conference

proceedings

2007 USENIX Annual Technical Conference

Santa Clara, CA, USA

June 17–22, 2007

Sponsored by

The USENIX Association

USENIX

Proceedings of the 2007 USENIX Annual Technical Conference

Santa Clara, CA, USA, June 17–22, 2007

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • http://www.usenix.org

The price is $45 for members and $55 for nonmembers.
Outside the U.S.A. and Canada, please add
$20 per copy for postage (via air printed matter).

Thanks to Our Sponsors

Thanks to Our Media Sponsors

ACM *Queue*
Addison-Wesley Professional/
Prentice Hall Professional
Dr. Dobb's Journal
Free Software Magazine
GRIDtoday
HPCwire
IDG World Expo
IEEE Security & Privacy
ITtoolbox
Linux Journal
Linux Pro Magazine
No Starch Press
OSTG
Penton TechX
SNIA
StorageNetworking.org
Sys Admin
UserFriendly.org

© 2007 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-931971-53-6

JEFF MOGUL

USENIX Association

Proceedings of the
2007 USENIX Annual Technical Conference

June 17–22, 2007
Santa Clara, CA, USA

Conference Organizers

Program Chairs

Jeff Chase, *Duke University*
Srinivasan Seshan, *Carnegie Mellon University*

Program Committee

Atul Adya, *Microsoft Research*
Matt Blaze, *University of Pennsylvania*
George Candea, *EPFL*
Miguel Castro, *Microsoft Research, Cambridge*
Fay Chang, *Google*
Nick Feamster, *Georgia Institute of Technology*
Marc Fiuczynski, *Princeton University/PlanetLab*
Terence Kelly, *Hewlett-Packard Labs*
Eddie Kohler, *University of California, Los Angeles, and Mazu Networks*
Z. Morley Mao, *University of Michigan*
Erich Nahum, *IBM T.J. Watson Research Center*
Jason Nieh, *Columbia University and VMware*
Brian Noble, *University of Michigan*
Timothy Roscoe, *ETH Zürich*
Emin Gün Sirer, *Cornell University*
Mike Swift, *University of Wisconsin, Madison*
Renu Tewari, *IBM Almaden Research Center*
Win Treese, *SiCortex, Inc.*
Andrew Warfield, *Cambridge University and XenSource*
Matt Welsh, *Harvard University*
Yuanyuan Zhou, *University of Illinois at Urbana-Champaign*

Poster Session Chair

Mike Swift, *University of Wisconsin, Madison*

Invited Talks Committee

Daniel V. Klein, *USENIX*
Ellie Young, *USENIX*

Guru Is In Coordinators

Daniel V. Klein, *USENIX*
Mary Seabrook, *Consultant*

The USENIX Association Staff

External Reviewers

Katerina Argyraki
Andrea Arpaci-Dusseau
Mary Baker
Bill Bolosky
Landon Cox
Brendan Cully
John Douceur
Fred Douglis
John Dunagan
Kave Eshghi
Craig Everhart
Michael Fetterman
Jason Flinn
Keir Fraser
Tal Garfinkel
Steven Hand
Tim Harris
Dean Hildebrand
Jon Howell
Galen Hunt
Emre Kıcıman
Tadayoshi Kohno
Christian Kreibich
Geoffrey Lefebvre
Xue Liu
Jacob Lorch
Varun Marupadi
Arif Merchant
Mark Miller
Manoj Naik
Dushyanth Narayanan
Dina Papagiannaki
Yaoping Ruan
Piyush Shivam
Sharad Singhal
Dan Sorin
Christopher Stewart
Yin Wang
Alec Wolman
Charles P. Wright
Gala Yadgar
Haifeng Yu

2007 USENIX Annual Technical Conference
June 17–22, 2007
Santa Clara, CA, USA

Index of Authors vi

Message from the Program Chairs vii

Wednesday, June 20

Tricks with Virtual Machines

Energy Management for Hypervisor-Based Virtual Machines 1
Jan Stoess, Christian Lang, and Frank Bellosa, University of Karlsruhe, Germany

Xenprobes, a Lightweight User-Space Probing Framework for Xen Virtual Machine 15
Nguyen Anh Quynh and Kuniyasu Suzaki, National Institute of Advanced Industrial Science and Technology, Japan

Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache 29
Pin Lu and Kai Shen, University of Rochester

Network Monitoring and Management

Hyperion: High Volume Stream Archival for Retrospective Querying 45
Peter J. Desnoyers and Prashant Shenoy, University of Massachusetts

Load Shedding in Network Monitoring Applications 59
Pere Barlet-Ros, Technical University of Catalonia; Gianluca Iannaccone, Intel Research Berkeley; Josep Sanjuàs-Cuxart, Diego Amores-López, and Josep Solé-Pareta, Technical University of Catalonia

Configuration Management at Massive Scale: System Design and Experience 73
William Enck and Patrick McDaniel, Pennsylvania State University; Subhabrata Sen, Panagiotis Sebos, and Sylke Spoerel, AT&T Research; Albert Greenberg, Microsoft Research; Sanjay Rao, Purdue University; William Aiello, University of British Columbia

Programming Abstractions for Network Services

Events Can Make Sense 87
Maxwell Krohn, MIT CSAIL; Eddie Kohler, University of California, Los Angeles; M. Frans Kaashoek, MIT CSAIL

MapJAX: Data Structure Abstractions for Asynchronous Web Applications 101
Daniel S. Myers, Jennifer N. Carlisle, James A. Cowling, and Barbara H. Liskov, MIT CSAIL

Sprockets: Safe Extensions for Distributed File Systems 115
Daniel Peek, Edmund B. Nightingale, and Brett D. Higgins, University of Michigan; Puspesh Kumar, IIT Kharagpur; Jason Flinn, University of Michigan

Thursday, June 21

Distributed Storage

SafeStore: A Durable and Practical Storage System 129
Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin, The University of Texas at Austin

POTSHARDS: Secure Long-Term Storage Without Encryption 143
Mark W. Storer, Kevin M. Greenan, and Ethan L. Miller, University of California, Santa Cruz; Kaladhar Voruganti, Network Appliance

Dandelion: Cooperative Content Distribution with Robust Incentives 157
Michael Sirivianos, Jong Han Park, Xiaowei Yang, and Stanislaw Jarecki, University of California, Irvine

Data and Indexing

Using Provenance to Aid in Personal File Search 171
Sam Shah, University of Michigan; Craig A.N. Soules, HP Labs; Gregory R. Ganger, Carnegie Mellon University; Brian D. Noble, University of Michigan

Supporting Practical Content-Addressable Caching with CZIP Compression 185
KyoungSoo Park and Sunghwan Ihm, Princeton University; Mic Bowman, Intel Research; Vivek S. Pai, Princeton University

Short Paper: Implementation and Performance Evaluation of Fuzzy File Block Matching 199
Bo Han and Pete Keleher, University of Maryland, College Park

System Security

From Trusted to Secure: Building and Executing Applications That Enforce System Security 205
Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel, Pennsylvania State University

From STEM to SEAD: Speculative Execution for Automated Defense 219
Michael E. Locasto, Angelos Stavrou, Gabriela F. Cretu, and Angelos D. Keromytis, Columbia University

Dynamic Spyware Analysis 233
Manuel Egele, Christopher Kruegel, and Engin Kirda, Secure Systems Lab, Technical University Vienna; Heng Yin, Carnegie Mellon University and College of William and Mary; Dawn Song, Carnegie Mellon University

Close to the Hardware

Evaluating Block-level Optimization Through the IO Path 247
Alma Riska, Seagate Research; James Larkby-Lahet, University of Pittsburgh; Erik Riedel, Seagate Research

DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch 261
Xiaoning Ding, Ohio State University; Song Jiang, Wayne State University; Feng Chen, Ohio State University; Kei Davis, Los Alamos National Laboratory; Xiaodong Zhang, Ohio State University

Short Paper: A Memory Soft Error Measurement on Production Systems 275
Xin Li, Kai Shen, and Michael C. Huang, University of Rochester; Lingkun Chu, Ask.com

Friday, June 22

Networked Systems

Addressing Email Loss with SureMail: Measurement, Design, and Evaluation 281
Sharad Agarwal and Venkata N. Padmanabhan, Microsoft Research; Dilip A. Joseph, University of California, Berkeley

Wresting Control from BGP: Scalable Fine-Grained Route Control 295
Patrick Verkaik, University of California, San Diego; Dan Pei, Tom Scholl, and Aman Shaikh, AT&T Labs—Research; Alex C. Snoeren, University of California, San Diego; Jacobus E. van der Merwe, AT&T Labs—Research

A Comparison of Structured and Unstructured P2P Approaches to Heterogeneous Random Peer Selection 309
Vivek Vishnumurthy and Paul Francis, Cornell University

Kernels

Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems 323
Oren Laadan and Jason Nieh, Columbia University

Reboots Are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly 337
Andrew Baumann, University of New South Wales and National ICT Australia; Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, and Orran Krieger, IBM T.J. Watson Research Center; Gernot Heiser, University of New South Wales and National ICT Australia

Short Paper: Exploring Recovery from Operating System Lockups 351
Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell, University of Illinois at Urbana-Champaign

Short Papers

Short Paper: Supporting Multiple OSes with OS Switching 357
Jun Sun, Dong Zhou, and Steve Longerbeam, DoCoMo USA Labs

Short Paper: Cool Job Allocation: Measuring the Power Savings of Placing Jobs at Cooling-Efficient Locations in the Data Center 363
Cullen Bash and George Forman, Hewlett-Packard Labs

Short Paper: Passwords for Everyone: Secure Mnemonic-based Accessible Authentication 369
Umut Topkara, Mercan Topkara, and Mikhail J. Atallah, Purdue University

Short Paper: Virtually Shared Displays and User Input Devices 375
Grant Wallace and Kai Li, Princeton University

Index of Authors

Agarwal, Sharad 281
Aiello, William 73
Alvisi, Lorenzo 129
Amores-López, Diego 59
Appavoo, Jonathan 337
Atallah, Mikhail J. 369
Barlet-Ros, Pere 59
Bash, Cullen 363
Baumann, Andrew 337
Bellosa, Frank. 1
Bowman, Mic 185
Campbell, Roy H. 351
Carlisle, Jennifer N. 101
Carlyle, Jeffrey C. 351
Chen, Feng 261
Chu, Lingkun 275
Cowling, James A. 101
Cretu, Gabriela F. 219
Da Silva, Dilma 337
Dahlin, Mike. 129
David, Francis M. 351
Davis, Kei 261
Desnoyers, Peter J. 45
Ding, Xiaoning 261
Egele, Manuel. 233
Enck, William. 73
Flinn, Jason. 115
Forman, George 363
Francis, Paul 309
Ganger, Gregory R. 171
Greenan, Kevin M. 143
Greenberg, Albert 73
Han, Bo. 199
Heiser, Gernot. 337
Hicks, Boniface 205
Higgins, Brett D. 115
Huang, Michael C. 275
Iannaccone, Gianluca 59
Ihm, Sunghwan. 185
Jaeger, Trent 205
Jarecki, Stanislaw 157
Jiang, Song 261
Joseph, Dilip A. 281
Kaashoek, M. Frans 87
Keleher, Pete. 199
Keromytis, Angelos D. 219
Kirda, Engin 233
Kohler, Eddie 87
Kotla, Ramakrishna 129
Krieger, Orran. 337
Krohn, Maxwell 87
Kruegel, Christopher 233
Kumar, Puspesh 115
Laadan, Oren 323
Lang, Christian 1
Larkby-Lahet, James 247
Li, Kai. 375
Li, Xin 275
Liskov, Barbara H. 101
Locasto, Michael E. 219
Longerbeam, Steve. 357
Lu, Pin 29
McDaniel, Patrick. 73, 205
Miller, Ethan L. 143
Myers, Daniel S. 101
Nieh, Jason 323
Nightingale, Edmund B. 115
Noble, Brian D. 171
Padmanabhan, Venkata N. 281
Pai, Vivek S. 185
Park, Jong Han 157
Park, KyoungSoo 185
Peek, Daniel 115
Pei, Dan 295
Quynh, Nguyen Anh. 15
Rao, Sanjay. 73
Riedel, Erik. 247
Riska, Alma 247
Rueda, Sandra. 205
Sanjuàs-Cuxart, Josep 59
Scholl, Tom. 295
Sebos, Panagiotis 73
Sen, Subhabrata 73
Shah, Sam. 171
Shaikh, Aman 295
Shen, Kai 29, 275
Shenoy, Prashant. 45
Sirivianos, Michael. 157
Snoeren, Alex C. 295
Solé-Pareta, Josep. 59
Song, Dawn 233
Soules, Craig A.N. 171
Spoerel, Sylke. 73
Stavrou, Angelos. 219
Stoess, Jan 1
Storer, Mark W. 143
Sun, Jun 357
Suzaki, Kuniyasu 15
Topkara, Mercan. 369
Topkara, Umut 369
van der Merwe, Jacobus E. 295
Verkaik, Patrick 295
Vishnumurthy, Vivek 309
Voruganti, Kaladhar 143
Wallace, Grant 375
Wisniewski, Robert W. 337
Yang, Xiaowei 157
Yin, Heng 233
Zhang, Xiaodong 261
Zhou, Dong. 357

Message from the Program Chairs

Welcome to Santa Clara for the 2007 USENIX Annual Technical Conference! We are proud of this year's program, which continues the USENIX tradition of technical excellence firmly grounded in the practice of computing.

As always, the outstanding and dedicated staff of USENIX have done most of the hard work to make this conference a success. It is always a pleasure to work with them, and we should all thank them for the service they provide for the community. Ellie Young is a force of nature, and the production staff have pulled all of the various inputs into a professional program. Special thanks as always to Jane-Ellen Long and also to Jennifer Joost, Anne Dickison, Casey Henderson, Devon Shaw, and all of the other staff working behind the scenes. Where they had to depend on us, we thank them for their patience and understanding.

We received 117 paper submissions for the refereed technical program this year: 102 as full submissions and 15 as short-paper submissions. We accepted 24 full papers and 7 short papers. The reviewing process generated 498 reviews: each PC member reviewed 21 papers, with additional reviews from 42 external reviewers. The Program Committee made the final decisions in a ten-hour meeting hosted at CMU. The Program Committee and reviewers did a great job working under pressure to complete reviewing and get the program out in little over six weeks. We thank all of them for their dedication and effort in providing useful feedback to the authors and maintaining strong quality standards for the program. Special thanks to Eddie Kohler for writing the new reviewing system we used this year, and to Joe Shamblin and Richard Braun for hosting it at Duke.

In addition, Dan Klein and Ellie Young have assembled a great set of invited speakers, with a mix of fresh and familiar faces and a wide range of exciting and important topics. There is also a poster session organized by Mike Swift.

We would like to thank our industry sponsors for making this event possible, most notably Google, Berkeley Communications, HP, and VMware.

Jeff Chase
Srinivasan Seshan
Program Chairs

Energy Management for Hypervisor-Based Virtual Machines

Jan Stoess Christian Lang Frank Bellosa
System Architecture Group, University of Karlsruhe, Germany
{stoess, chlang, bellosa}@ira.uka.de

Abstract

Current approaches to power management are based on operating systems with full knowledge of and full control over the underlying hardware; the distributed nature of multi-layered virtual machine environments renders such approaches insufficient. In this paper, we present a novel framework for energy management in modular, multi-layered operating system structures. The framework provides a unified model to partition and distribute energy, and mechanisms for energy-aware resource accounting and allocation. As a key property, the framework explicitly takes the recursive energy consumption into account, which is spent, e.g., in the virtualization layer or subsequent driver components.

Our prototypical implementation targets hypervisor-based virtual machine systems and comprises two components: a host-level subsystem, which controls machine-wide energy constraints and enforces them among all guest OSes and service components, and, complementary, an energy-aware guest operating system, capable of fine-grained application-specific energy management. Guest level energy management thereby relies on effective virtualization of physical energy effects provided by the virtual machine monitor. Experiments with CPU and disk devices and an external data acquisition system demonstrate that our framework accurately controls and stipulates the power consumption of individual hardware devices, both for energy-aware and energy-unaware guest operating systems.

1 Introduction

Over the past few years, virtualization technology has regained considerable attention in the design of computer systems. Virtual machines (VMs) establish a development path for incorporating new functionality – server consolidation, transparent migration, secure computing, to name a few – into a system that still retains compatibility to existing operating systems (OSes) and applications. At the very same time, the ever increasing power density and dissipation of modern servers has turned energy management into a key concern in the design of OSes. Research has proposed several approaches to OS directed control over a computer's energy consumption, including user- and service-centric management schemes. However, most current approaches to energy management are developed for standard, legacy OSes with a monolithic kernel. A monolithic kernel has full control over all hardware devices and their modes of operation; it can directly regulate device activity or energy consumption to meet thermal or energy constraints. A monolithic kernel also controls the whole execution flow in the system. It can easily track the power consumption at the level of individual applications and leverage its application-specific knowledge during device allocation to achieve dynamic and comprehensive energy management.

Modern VM environments, in contrast, consist of a distributed and multi-layered software stack including a hypervisor, multiple VMs and guest OSes, device driver modules, and other service infrastructure (Figure 1). In such an environment, direct and centralized energy management is unfeasible, as device control and accounting information are distributed across the whole system.

At the lowest-level of the virtual environment, the privileged hypervisor and host driver modules have direct control over hardware devices and their energy consumption. By inspecting internal data structures, they can obtain coarse-grained per-VM information on how energy is spent on the hardware. However, the host level does not possess any knowledge of the energy consumption of individual applications. Moreover, with the ongoing trend to restrict the hypervisor's support to a minimal set of hardware and to perform most of the device control in unprivileged driver domains [8,15], hypervisor and driver modules each have direct control over a small set of devices; but they are oblivious to the ones not managed by themselves.

The guest OSes, in turn, have intrinsic knowledge of their own applications. However, guest OSes operate on deprivileged virtualized devices, without direct access to the physical hardware, and are unaware that the hardware may be shared with other VMs. Guest OSes are also unaware of the *side-effects* on power consumption caused by the vir-

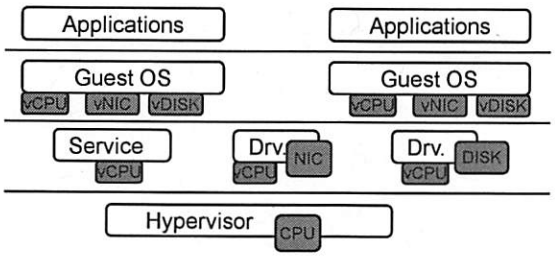


Figure 1: Increasing number of layers and components in today's virtualization-based OSes.

tual device logic: since virtualization is transparent, the "hidden", or recursive power consumption, which the virtualization layer itself causes when requiring the CPU or other resources simply vanishes unaccounted in the software stack. Depending on the complexity of the interposition, resource requirements can be substantial: a recent study shows that the virtualization layer requires a considerable amount of CPU processing time for I/O virtualization [5].

The whole situation is even worsened by the non-partitionability of some of the physical effects of power dissipation: the temperature of a power consuming device, for example, cannot simply be partitioned among different VMs in a way that each one gets alloted its own share on the temperature. Beyond the lack of comprehensive control over and knowledge of the power consumption in the system, we can thus identify the lack of a model to comprehensively express physical effects of energy consumption in distributed OS environments.

To summarize, current power management schemes are limited to legacy OSes and unsuitable for VM environments. Current virtualization solutions disregard most energy-related aspects of the hardware platform; they usually virtualize a set of standard hardware devices only, without any special power management capabilities or support for energy management. Up to now, power management for VMs is limited to the capabilities of the host OS in hosted solutions and mostly dispelled from the server-oriented hypervisor solutions.

Observing these problems, we present a novel framework for managing energy in distributed, multi-layered OS environments, as they are common in today's computer systems. Our framework makes three contributions. The first contribution is a model for partitioning and distributing energy effects; our model solely relies on the notion of energy as the base abstraction. Energy quantifies the physical effects of power consumption in a distributable way and can be partitioned and translated from a global, system-wide notion into a local, component- or user-specific one. The second contribution is a distributed energy accounting approach, which accurately tracks back the energy spent in the system to originating activities. In particular, the presented approach incorporates both the direct and the side-effectual energy consumption spent in the virtualization layers or subsequent driver components. As the third contribution, our framework exposes all resource allocation mechanisms from drivers and other resource managers to the respective energy management subsystems. Exposed allocation enables dynamic and remote regulation of energy consumption in a way that the overall consumption matches the desired constraints.

We have implemented a prototype that targets hypervisor-based systems. We argue that virtual server environments benefit from energy management within and across VMs; hence the prototype employs management software both at host-level and at guest-level. A host-level management subsystem enforces system-wide energy constraints among all guest OSes and driver or service components. It accounts direct and hidden power consumption of VMs and regulates the allocation of physical devices to ensure that each VM does not consume more than a given power allotment. Naturally, the host-level subsystem performs independent of the guest operating system; on the downside, it operates at low level and in coarse-grained manner. To benefit from fine-grained, application-level knowledge, we have complemented the host-level part with an optional energy-aware guest OS, which redistributes the VM-wide power allotments among its own, subordinate applications. In analogy to the host-level, where physical devices are allocated to VMs, the guest OS regulates the allocation of virtual devices to ensure that its applications do not spend more energy than their alloted budget.

Our experiments with CPU and disk devices demonstrate that the prototype effectively accounts and regulates the power consumption of individual physical and virtual devices, both for energy-aware and energy-unaware guest OSes.

The rest of the paper is structured as follows: In Section 2, we present a generic model to energy management in distributed, multi-layered OS environments. We then detail our prototypical implementation for hypervisor-based systems in Section 3. We present experiments and results in Section 4. We then discuss related approaches in Section 5, and finally draw a conclusion and outline future work in Section 6.

2 Distributed Energy Management

The following section presents the design principles we consider to be essential for distributed energy management. We begin with formulating the goals of our work. We then describe the unified energy model that serves as a foundation for the rest of our approach. We finally describe the overall structure of our distributed energy management framework.

2.1 Design Goals

The increasing number of layers, components, and subsystems in modern OS structures demands for a distributed approach to control the energy spent in the system. The approach must perform effectively across protection boundaries, and it must comprise different types of activities, software abstractions, and hardware resources. Furthermore, the approach must be flexible enough to support diversity in energy management paradigms. The desire to control power and energy effects of a computer system stems from a variety of objectives: Failure rates typically increase with the temperature of a computer node or device; reliability requirements or limited cooling capacities thus directly translate into *temperature constraints*, which are to be obeyed for the hardware to operate correctly. Specific *power limits*, in turn, are typically imposed by battery or backup generators, or by contracts with the power supplier. Controlling power consumption on a *per-user base* finally enables accountable computing, where customers are billed for the energy consumed by their applications, but also receive a guaranteed level or quality of service. However, not only the objectives for power management are diverse; there also exists a variety of algorithms to achieve those objectives. Some of them use real temperature sensors, whereas others rely on estimation models [3, 12]. To reach their goals, the algorithms employ different mechanisms, like throttling resource usage, request batching, or migrating of execution [4, 9, 17]. Hence, a valid solution must be flexible and extensible enough to suit a diversity of goals and algorithms.

2.2 Unified Energy Model

To encompass the diverse demands on energy management, we propose to use the notion of energy as the base abstraction in our system, an approach which is similar to the currentcy model in [28]. The key advantage of using energy is that it quantifies power consumption in a partitionable way – unlike other physical effects of power consumption such as the temperature of a device. Such effects can easily be expressed as energy constraints, by means of a thermal model [3, 12]. The energy constraints can then be partitioned from global notions into local, component-wise ones. Energy constraints also serve as a coherent base metric to unify and integrate management schemes for different hardware devices.

2.3 Distributed Management

Current approaches to OS power management are tailored to single building-block OS design, where one kernel instance manages all software and hardware resources. We instead model the OS as a set of components, each responsible for controlling a hardware device, exporting a service library, or providing a software resource for use by applications.

Our design is guided by the familiar concept of separating policy and mechanism. We formulate the procedure of energy management as a simple feedback loop: the first step is to determine the current power consumption and to account it to the originating activities. The next step is to analyze the accounting data and to make a decision based on a given policy or goal. The final step is to respond with allocation or de-allocation of energy consuming resources to the activities, with the goal to align the energy consumption with the desired constraints.

We observe that mainly the second step is associated with policy, whereas the two other steps are mechanisms, bound to the respective providers of the resource, which we hence call *resource drivers*. We thus model the second step as an *energy manager module*, which may, but need not reside in a separate software component or protection domain. Multiple such managers may exist concurrently the system, at different position in the hierarchy and with different scopes.

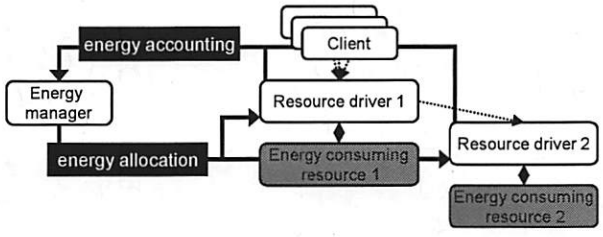


Figure 2: Distributed energy management. Energy managers may reside in different components or protection domains. Resource drivers consume resources themselves, for which the energy is accounted back to the original clients.

Each energy manager is responsible for a set of subordinate resources and their energy consump-

tion. Since the system is distributed, the resource manager cannot assume direct control or access over the resource; it requires remote mechanisms to account and allocate the energy (see Figure 2). Hence, by separating policy from mechanism, we translate our general goal of distributed energy management into the two specific aspects of *distributed energy accounting* and *dynamic, exposed resource allocation*; these are the subject of the following paragraphs.

Distributed energy accounting Estimating and accounting the energy of a physical device usually requires detailed knowledge of the particular device. Our framework therefore requires each driver of an energy consuming device or resource to be capable of determining (or estimating) the energy consumption of its resources. Likewise, it must be capable to account the consumption to its consumers. If the energy management software resides outside the resource driver, it must propagate the accounting information to the manager.

Since the framework does not assume a single kernel comprising all resource subsystems, it has to track energy consumptions across module boundaries. In particular, it must incorporate the recursive energy consumption: that is, the driver of a given resource such as a disk typically requires other resources, like the CPU, in order to provide its service successfully. Depending on the complexity, such recursive resource consumption may be substantial; consider, as examples, a disk driver that transparently encrypts and decrypts its client requests, or a driver that forwards client requests to a network attached storage server via a network interface card. Recursive resource consumption requires energy, which must be accounted back to the clients. In our example, it would be the responsibility of disk driver to calculate its clients' shares of the disk and on its own CPU energy. To determine its CPU energy, the driver must recursively query the driver of the CPU resource, which is the hypervisor in our case.

Dynamic and exposed resource allocation To regulate the energy spent on a device or resource, each driver must expose its allocation mechanisms to energy manager subsystems. The manager leverages the allocation mechanisms to ensure that energy consumption matches the desired constraints. Allocation mechanisms relevant for energy management can be roughly distinguished into hardware and software mechanisms. Hardware-provided power saving features typically provide a means to change power consumption of a device, by offering several modes of operation with different efficiency and energy coefficients (e.g., halt cycles or different active and sleep modes). The ultimate goal is to achieve the optimal level of efficiency with respect to the current resource utilization, and to reduce the wasted power consumption. Software-based mechanisms, in turn, rely on the assumption that energy consumption depends on the level of utilization, which is ultimately dictated by the number of device requests. The rate of served requests can thus be adapted by software to control the power consumption.

3 A Prototype for Hypervisor-Based Systems

Based on the design principles presented above, we have developed a distributed, two-level energy management framework for hypervisor-based VM systems. The prototype employs management software both at host-level and at guest-level. It currently supports management of two main energy consumers, CPU and disk. CPU services are directly provided by the hypervisor, while the disk is managed by a special device driver VM. In the following section, we first describe the basic architecture of our prototype. We then present the energy model for CPU and disk devices. We then describe the host-level part, and finally the guest-level part of our energy management prototype.

3.1 Prototype Architecture

Our prototype uses the L4 micro-kernel as the privileged hypervisor, and para-virtualized Linux kernel instances running on top of it. L4 provides core abstractions for user level resource management: virtual processors (kernel threads), synchronous communication, and mechanisms to recursively construct virtual address spaces. I/O devices are managed at user-level; L4 only deals with exposing interrupts and providing mechanisms to protect device memory.

The guest OSes are adaptions of the Linux 2.6 kernel, modified to run on top of L4 instead of on bare hardware [11]. For managing guest OS instances, the prototype includes a user-level VM monitor (VMM), which provides the virtualization service based on L4's core abstractions. To provide user-level device driver functionality, the framework dedicates a special device driver VM to each device, which exports a virtual device interface to client VMs and multiplexes virtual device requests onto the physical device. The driver VMs are Linux guest OS instances

themselves, which encapsulate and reuse standard Linux device driver logic for hardware control [15].

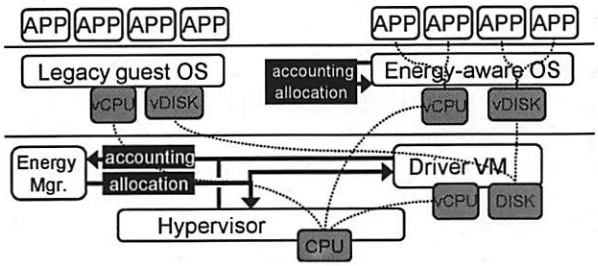


Figure 3: Prototype architecture. The host-level subsystem controls system-wide energy constraints and enforces them among all guests. A complementary energy-aware guest OS is capable of performing its own, application-specific energy management.

The prototype features a host-level energy manager module responsible for controlling the energy consumption of VMs on CPUs and disk drives. The energy manager periodically obtains the per-VM CPU and disk energy consumption from the hypervisor and driver VM, and matches them against a given power limit. To bring both in line, it responds by invoking the exposed throttling mechanisms for the CPU and disk devices. Our energy-aware guest OS is a modified version of L4Linux that implements the resource container abstraction [1] for resource management and scheduling. We enhanced the resource containers to support energy management of virtual CPUs and disks. Since the energy-aware guest OS requires virtualization of the energy effects of CPU and disk, the hypervisor and driver VM propagate their accounting records to the user-level VM monitor. The monitor then creates, for each VM, a local view on the current energy consumption, and thereby enables the guest to pursue its own energy-aware resource management. Note, that our energy-aware guest OS is an optional part of the prototype: it provides the benefit of fine-grained energy management for Linux-compatible applications. For all energy-unaware guests, our prototype resorts to the coarser-grained host-level management, which achieves the constraints regardless whether the guest-level subsystem is present or not.

Figure 3 gives a schematic overview of the basic architecture. Our prototype currently runs on IA-32 microprocessors. Certain parts, like the device driver VMs, are presently limited to single processor systems; we are working on multi-processor support and will integrate it into future versions.

3.2 Device Energy Models

In the following section, we present the device energy models that serve as a base for CPU and disk accounting. We generally break down the energy consumption into *access* and *idle consumption*. Access consumption consists of the energy spent when using the device. This portion of the energy consumption can be reduced by controlling device allocation, e.g., in terms of the client request rate. Idle consumption, in turn, is the minimum power consumption of the device, which it needs even when it does not serve requests. Many current microprocessors support multiple sleep and active modes, e.g., via frequency scaling or clock gating. A similar technology, though not yet available on current standard servers, can be found in multi-speed disks, which allow lowering the spinning speed during phases of low disk utilization [10]. To retain fairness, we propose to decouple the power state of a multi-speed device from the accounting of its idle costs. Clients that do not use the device are charged for the lowest active power state. Higher idle consumptions are only charged to the clients are actively using the device.

3.2.1 CPU Energy Model

Our prototype leverages previous work [3, 13] and bases CPU energy estimation on the rich set of performance counters featured by modern IA-32 microprocessors. For each performance counter event, the approach assigns a weight representing its contribution to the processor energy. The weights are the result of a calibration procedure that employs test applications with constant and known power consumptions and physical instrumentation of the microprocessors [3]. Previous experiments have demonstrated that this approach is fairly accurate for integer applications, with an error of at most 10 percent. To obtain the processor energy consumption during a certain period of time, e.g., during execution of a VM, the prototype sums up the number of events that occurred during that period, multiplied with their weights. The time stamp counter, which counts clock cycles regardless whether the processor is halted or not, yields an accurate estimation of the CPU's idle consumption.

3.2.2 Disk Energy Model

Our disk energy model differs from the CPU model in that it uses a time-based approach rather than event sampling. Instead of attributing energy consumption to events, we attribute power consumption to different device states, and calculate the time the

device requires to transfer requests of a given size. There is no conceptual limit to the number of power states. However, we consider suspending the disk to be an unrealistic approach for hypervisor systems; for lack of availability, we do not consider multi-speed disks as well. We thus distinguish two different power states: active and idle.

To determine the transfer time of a request – which is equal to the time the device must remain in active state to handle it –, we divide the size of the request by the disk's transfer rate in bytes per second. We calculate the disk transfer rate dynamically, in intervals of 50 requests. Although we ignore several parameters that affect the energy consumption of requests, (e.g., seek time or the rotational delays), our evaluation shows that our simple approach is sufficiently accurate. Our observation is substantiated by the study in [26], which indicates that such a 2-parameter model is inaccurate only because of sleep-modes, which we can safely disregard for our approach.

3.3 Host-Level Energy Management

Our framework requires each driver of a physical device to determine the device's energy consumption and to account the consumption to the client VMs. The accounting infrastructure uses the device energy model presented above: Access consumption is charged directly to each request, after the request has been fulfilled. The idle consumption, in turn, cannot be attributed to specific requests; rather, it is alloted to all client VMs in proportion to their respective utilization. For use by the energy manager and others, the driver grants access to its accounting records via shared memory and updates the records regularly.

In addition to providing accounting records, each resource driver exposes its allocation mechanisms to energy managers and other resource management subsystems. At host-level, our framework currently supports two allocation mechanisms: CPU throttling and disk request shaping. CPU throttling can be considered as a combined software-hardware approach, which throttles activities in software and spends the unused time in halt cycles. Our disk request shaping algorithm is implemented in software.

In the remainder of this section, we first explain how we implemented runtime energy accounting and allocation for CPU and the disk devices. We then detail how the these mechanisms enable our energy management software module to keep the VMs' energy consumption within constrained limits.

3.3.1 CPU Energy Accounting

To accurately account the CPU energy consumption, we trace the performance counter events within the hypervisor and propagate them to the user-space energy manager module. Our approach extends our previous work to support resource management via event logging [20] to the context of energy management. The tracing mechanism instruments context switches between VMs within the hypervisor; at each switch, it records the current values of the performance counters into an in-memory log buffer. The hypervisor memory-maps the buffers into the address space of the energy manager. The energy manager periodically analyzes the log buffer and calculates the energy consumption of each VM (Figure 4).

By design, our tracing mechanism is asynchronous and separates performance counter accumulation from their analysis and the derivation of the energy consumption. It is up to the energy manager to perform the analysis often enough to ensure timeliness and accuracy. Since the performance counter logs are relatively small, we consider this to be easy to fulfil; our experience shows that the performance counter records cover a few hundred or thousand bytes, if the periodical analysis is performed about every 20th millisecond.

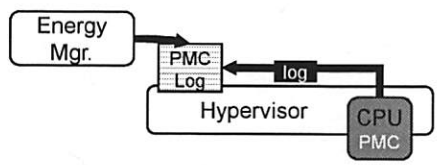


Figure 4: The hypervisor collects performance counter traces and propagates the trace logs to user-space energy managers.

The main advantage of using tracing for recording CPU performance counters is that it separates policy from mechanism. The hypervisor is extended by a simple and cheap mechanism to record performance counter events. All aspects relevant to energy estimation and policy are kept outside the hypervisor, within the energy manager module. A further advantage of in-memory performance counter records is that they can easily be shared – propagating them to other guest-level energy accountants is a simple matter of leveraging the hypervisor's memory-management primitives.

In the current prototype, the energy manager is invoked every 20 ms to check the performance counter logs for new records. The log records contain the performance counter values relevant for en-

ergy accounting, sampled at each context switch together with an identifier of the VM that was active on the CPU. For each period between subsequent context switches, the manager calculates the energy consumption during that period, by multiplying the advance of the performance counters with their weights. Rather than charging the complete energy consumption to the active VM, the energy manager subtracts the idle cost and splits it between all VMs running on that processor. The time stamp counter, which is included in the recorded performance counters, provides an accurate estimation of the processor's idle cost. Thus the energy estimation looks as follows:

```
/* per-VM idle energy based on TSC advance (pmc0) */
for (id = 0; id < max_vms; id++)
   vm[id].cpu_idle += weight[0] * pmc[0] / max_vms;

/* calculate and charge access energy (pmc1..pmc8) */
for (p=1; p < 8; p++)
  vm[cur_id].cpu_access += weight[p] * pmc[p];
```

3.3.2 Disk Energy Accounting

To virtualize physical disks drives, our framework reuses legacy Linux disk driver code by executing it inside VMs. The driver functionality is exported via a translation module that mediates requests between the device driver and external client VMs. The translation module runs in the same address space as the device driver and handles all requests sent to and from the driver. It receives disk requests from other VMs, translates them to basic Linux block I/O requests, and passes them to the original device driver. When the device driver has finalized the request, the module again translates the result and returns it to the client VM.

The translation module has access to all information relevant for accounting the energy dissipated by the associated disk device. We implemented accounting completely in this translation module, without changing the original device driver. The module estimates the energy consumption of the disk using the energy model presented above. When the device driver has completed a request, the translation module estimates the energy consumption of the request, depending on the number of transferred bytes:

```
/* estimate transfer cost for size bytes */
vm[cur_id].disk_access += (size / transfer_rate)
          * (active_disk_power - idle_disk_power);
```

Because the idle consumption is independent of the requests, it does not have to be calculated for each request. However, the driver must recalculate it periodically, to provide the energy manager with up-to-date accounting records power consumption of the disk. For that purpose, the driver invokes the following procedure periodically every 50 ms:

```
/* estimate idle energy since last time */
idle_disk_energy = idle_disk_power * (now - last)
                   / max_client_vms;
for (id = 0; v < max_client_vms; id++)
   vm[id].disk_idle += idle_disk_energy;
```

3.3.3 Recursive Energy Accounting

Fulfilling a virtual device request issued by a guest VM may involve interacting with several different physical devices. Thus, with respect to host-level energy accounting, it is not sufficient to focus on single physical devices; rather, accounting must incorporate the energy spent recursively in the virtualization layer or subsequent service.

We therefore perform a recursive, request-based accounting of the energy spent in the system, according to the design principles presented in Section 2. In particular, each driver of a physical device determines the energy spent for fulfilling a given request and passes the cost information back to its client. If the driver requires other devices to fulfill a request, it charges the additional energy to its clients as well. Since idle consumption of a device cannot be attributed directly to requests, each driver additionally provides an "electricity meter" for each client. It indicates the client's share in the total energy consumption of the device, including the cost already charged with the requests. A client can query the meter each time it determines the energy consumption of its respective clients.

As a result, recursive accounting yields a distributed matrix of virtual-to-physical transactions, consisting of the idle and the active energy consumption of each physical device required to provide a given virtual device (see Figure 5). Each device driver is responsible for reporting its own vector of the physical device energy it consumes to provide its virtual device abstraction.

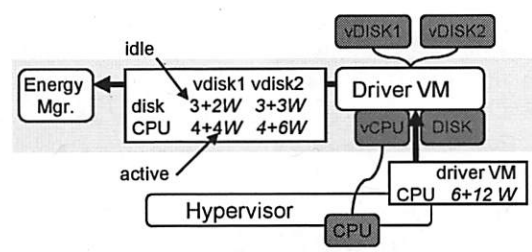


Figure 5: Recursive accounting of disk energy consumption; for each client VM and physical device, the driver reports idle and active energy to the energy manager. The driver is assumed to consume 8W CPU idle power, which is apportioned equally to the two clients.

Since our framework currently supports CPU and disk energy accounting, the only case where recursive accounting is required occurs in the virtual disk driver located in the driver VM. The cost for the virtualized disk consists of the energy consumed by the disk and the energy consumed by the CPU while processing the requests. Hence, our disk driver also determines the processing energy for each request in addition to the disk energy as presented above.

As with disk energy accounting, we instrumented the translation module in the disk driver VM to determine the active and idle CPU energy per client VM. The Linux disk driver combines requests to get better performance and delays part of the processing in work-queues and tasklets. When determining the active CPU energy, it would be infeasible to track the CPU energy consumption of each individual request. Instead, we retrieve the CPU energy consumption at times and apportion it between the requests. Since the driver runs in a VM, it relies on the energy virtualization capabilities of our framework to retrieve a local view on the CPU energy consumption (details on energy virtualization are presented in Section 3.4).

The Linux kernel constantly consumes a certain amount of energy, even if it does not handle disk requests. According to our energy model, we do not charge idle consumption with the request. To be able to distinguish the idle driver consumption from the access consumption, we approximate the idle consumption of the Linux kernel when no client VM uses the disk.

To account active CPU consumption, we assume constant values per request, and predict the energy consumption of future requests based on the past. Every 50th request, we estimate the driver's CPU energy consumption by means of virtualized performance monitoring counters and adjust the expected cost for the next 50 requests. The following code illustrates how we calculate the cost per request. In the code fragment, the static variable `unaccounted_cpu_energy` keeps track of the deviation between the consumed energy and the energy consumption already charged to the clients. The function `get_cpu_energy()` returns the guest-local view of the current idle and active CPU energy since the last query.

```
/* subtract idle CPU consumption of driver VM */
unaccounted_cpu_energy -= drv_idle_cpu_power
                          * (now - last);

/* calculate cost per request */
num_req = 50;
unaccounted_cpu_energy += get_cpu_energy();
unaccounted_cpu_energy -= cpu_req_energy * num_req;
cpu_req_energy = unaccounted_cpu_energy / num_req;
```

3.3.4 CPU Resource Allocation

To regulate the CPU energy consumption of individual machines, our hypervisor provides a mechanism to throttle the CPU allocation at runtime, from user-level. The hypervisor employs a stride scheduling algorithm [21,23] that allots proportional CPU shares to virtual processors; it exposes control over the shares to selected, privileged user-level components. The host-level energy manager dynamically throttles a virtual CPU's energy consumption by adjusting the alloted share accordingly. A key feature of stride scheduling is that it does not impose fixed upper bounds on CPU utilization: the shares have only relative meaning, and if one virtual processor does not fully utilize its share, the scheduler allows other, competing virtual processors to steal the unused remainder. An obvious consequence of dynamic upper bounds is that energy consumption will not be constrained either, at least not with a straight-forward implementation of stride scheduling. We solved this problem by creating a distinct and privileged *idle virtual processor* per CPU, which is guaranteed to spend all alloted time with issuing halt instructions (we modified our hypervisor to translate the idle processor's virtual halt instructions directly into real ones). Initially, each idle processor is alloted only a minuscule CPU share, thus all other virtual processors will be favored on the CPU if they require it. However, to constrain energy consumption, the energy manager will decrease the CPU shares of those virtual processors, and idle virtual processor will directly translate the remaining CPU time into halt cycles. Our approach guarantees that energy limits are effectively imposed; but it still preserves the advantageous processor stealing behavior for all other virtual processors. It also keeps the energy-policy out of the hypervisor and allows, for instance, to improve the scheduling policy with little effort, or to exchange it with a more throughput-oriented one for those environments where CPU energy management is not required.

3.3.5 Disk Request Shaping

To reduce disk power consumption, we pursue a similar approach and enable a energy manager to throttle disk requests of individual VMs. Throttling the request rate not only reduces the direct access consumption of the disk; it also reduces the recursive CPU consumption which the disk driver requires to process, recompute, and issue requests. We implemented the algorithm as follows: the disk driver processes a client VM's disk requests only to a specific request budget, and it delays all pending requests.

The driver periodically refreshes the budgets according to the specific throttling level set by the energy manager. The algorithm is illustrated by the following code snippet:

```
void process_io(client_t *client)
{
   ring = &client->ring;

   for (i=0; i < client->budget; i++)
   {
      desc = &client->desc[ ring->start ];
      ring->start = (ring->start+1) % ring->cnt;
      initiate_io(conn, desc, ring);
   }
}
```

3.3.6 Host-level Energy Manager

Our host-level energy manager manager relies on the accounting and allocation mechanisms described previously, and implements a simple policy that enforces given device power limits on a per-VM base. The manager consists of an initialization procedure and a subsequent feedback loop. During initialization, the manager determines a power limit for each VM and device type, which may not be exceeded during runtime. The CPU power limit reflects the active CPU power a VM is allowed to consume directly. The disk power limit reflects the overall active power consumption the disk driver VM is allowed to spend in servicing a particular VM, including the CPU energy spent for processing (Nevertheless, the driver's CPU and disk energy are accounted separately, as depicted by the matrix in Figure 5). Finding an optimal policy for allotment of power budgets is not the focus of our work; at present, the limits are set to static values.

The feedback loop is invoked periodically, every 100 ms for the CPU and every 200 ms for the disk. It first obtains the CPU and disk energy consumption of the past interval by querying the accounting infrastructure. The current consumptions are used to predict future consumptions. For each device, the manager compares the VM's current energy consumption with the desired power limit multiplied with the time between subsequent invocations. If they do not match for a given VM, the manager regulates the device consumption by recomputing and propagating the CPU strides and disk throttle factors respectively. To compute a new CPU stride, the manager adds or subtracts a constant offset from the current value. When computing the disk throttle factor, the manager takes the past period into consideration, and calculates the offset Δt according to the following formula. In this formula, e_c denotes the energy consumed, e_l the energy limit per period, and t and t_l and denote the present and past disk throttle factors; viable throttle factors range from 0 to a few thousand:

$$\Delta t = \begin{cases} \frac{1}{4}(t_l - t) + \frac{e_l - e_c}{|e_l - e_c|} & : \begin{cases} e_c > e_l, t > t_l \\ e_c < e_l, t < t_l \end{cases} \\ \frac{4}{3}(t - t_l) + \frac{e_l - e_c}{|e_l - e_c|} & : \quad else \end{cases}$$

3.4 Virtualized Energy

To enable application-specific energy management, our framework supports accounting and control not only for physical but also of virtual devices. In fact, the advantage of having guest-level support for energy accounting is actually twofold: first, it enables guest resource management subsystems to leverage their application-specific knowledge; second, it allows drivers and other components to recursively determine the energy for their services.

The main difference between a virtual device and other software services and abstractions lies in its interface: a virtual device closely resembles its physical counterpart. Unfortunately, most current hardware devices offer no direct way to query energy or power consumption. The most common approach to determine the energy consumption is to estimate it based on certain device characteristics, which are assumed to correlate with the power or energy consumption of the device. By emulating the according behavior for the virtual devices, we support energy estimation in the guest without major modifications to the guest's energy accounting. Our ultimate goal is to enable the guest to use the same driver for virtual and for real hardware. In the remainder of this section, we first describe how we support energy accounting of virtual CPU and disk. We then present the implementation of our energy-aware guest OS, which provides the support for application-specific energy management.

3.4.1 Virtual CPU Energy Accounting

For virtualization of physical energy effects of the CPU, we provide a virtual performance counter model that gives guest OSes a private view of their current energy consumption. The virtual model relies on the tracing of performance counters within the hypervisor, which we presented in Section 3.3.1. As mentioned, not only an energy-aware guest OS requires the virtual performance counters; the specialized device driver VM uses them as well, when recursively determining the CPU energy for its disk services.

Like their physical counterparts, each virtual CPU has a set of virtual performance counters, which

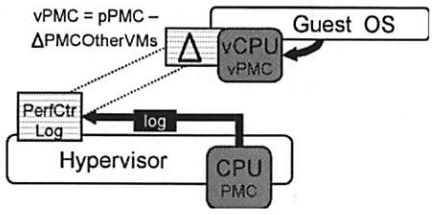


Figure 6: Virtualizing performance counters via hypervisor-collected performance counter traces.

factor out the events of other, simultaneously running VMs. If a guest OS determines the current value of a virtual performance counter, an emulation routine in the in-place monitor obtains the current hardware performance counter and subtracts all advances of the performance counters that occurred when other VMs were running. The hardware performance counters are made read-accessible to user-level software by setting a control register flag in the physical processors. The advances of other VMs are derived from the performance counter log buffers. To be accessible by the in-place VMM, the log buffers are mapped read-only into the address space of the guest OS.

3.4.2 Virtual Disk Energy Accounting

In contrast to the CPU, the disk energy estimation schemes does not rely on on-line measurements of sensors or counters; rather, it is based on known parameters such as the disk's power consumption in idle and active mode and the time it remains in active mode to handle a request. Directly translating the energy consumption of physical devices from our run-time energy model to the parameter-based model of the respective guest OS would yield only inaccurate results. The VMM would have to calibrate the energy consumption of the devices to calculate the energy parameters of the virtual devices. Furthermore, parameters of shared devices may change with the number of VMs, which contradicts the original estimation model. To ensure accuracy in the long run, the guest would have to query the virtual devices regularly for updated parameters.

For our current virtual disk energy model, we therefore use a para-virtual device extension. We expose each disk energy meter as an extension of the virtual disk device; energy-aware guest operating systems can take advantage of them by customizing the standard device driver appropriately.

3.4.3 An Energy-aware Guest OS

For application-specific energy management, we have incorporated the familiar resource container concept into a standard version of our para-virtualized Linux 2.6 adoption. Our implementation relies on a previous approach to use resource containers in the context of CPU energy management [3,24]. We extended the original version with support for disk energy. No further efforts were needed to manage virtual CPU energy; we only had to virtualize the performance counters to get the original version to run.

Similar to the host-level subsystem, the energy-aware guest operating system performs scheduling based on energy criteria. In contrast to standard schedulers, it uses resource containers as the base abstraction rather than threads or processes. Each application is assigned to a resource container, which then accounts all energy spent on its behalf. To account virtual CPU energy, the resource container implementation retrieves the (virtual) performance counter values on container switches, and charges the resulting energy to the previously active container. A container switch occurs on every context switch between processes residing in different containers.

To account virtual disk energy, we enhanced the client-side of the virtual device driver, which forwards disk requests to the device driver VM. Originally, the custom device driver received single disk requests from the Linux kernel, which contained no information about the user-level application that caused it. We added a pointer to a resource container to every data structure involved in a read or write operation. When an application starts a disk operation, we bind the current resource container to the according page in the page cache. When the kernel writes the pages to the virtual disk, we pass the resource container on to the respective data structures (i.e., buffer heads and bio objects). The custom device driver in the client accepts requests in form of bio objects and translates them to a request for the device driver VM. When it receives the reply together with the cost for processing the request, it charges the cost to the resource container bound to the bio structure.

To control the energy consumption of virtual devices, the guest kernel redistributes its own, VM-wide power limits to subordinate resource containers, and enforces them by means of preemption. Whenever a container exhausts the energy budget of the current period (presently set to 50 ms), it is preempted until a refresh occurs in the next period. A

simple user-level application retrieves the VM wide budgets from host-level energy-manager and passes them onto the guest kernel via special system calls.

4 Experiments and Results

In the following section, we present experimental results we obtained from our prototype. Our main goal is to demonstrate that our infrastructure provides an effective solution to manage energy in distributed, multi-layered OSes. We consider two aspects as relevant: At first, we validate the benefits of distributed energy accounting. We then present experiments that aim to show the advantages of multi-layered resource allocation to enforce energy constraints.

For CPU measurements, we used a Pentium D 830 with two cores at 3GHz. Since our implementation is currently limited to single processor systems, we enabled only on one core, which always ran at its maximum frequency. When idle, the core consumes about 42W; under full load, power consumption may be 100W and more. We performed disk measurements on a Maxtor DiamondMax Plus 9 IDE hard disk with 160GB size, for which we took the active power (about 5.6W) and idle power (about 3.6W) characteristics from the data sheet [16]. We validated our internal, estimation-based accounting mechanisms by means of an external high-performance data acquisition (DAQ) system, which measured the real disk and CPU power consumption with a sampling frequency of 1KHz.

4.1 Energy Accounting

To evaluate our approach of distributed energy accounting, we measured the overall energy required for using a virtual disk. For that purpose, we ran a synthetic disk stress test within a Linux guest OS. The test runs on a virtual hard drive, which is multiplexed on the physical disk by the disk driver VM. The test performs almost no computation, but generates heavy disk load. By opening the virtual disk in *raw* access mode, the test bypasses most of the guest OS's caching effects, and causes the file I/O to be performed directly to and from user space buffers. Afterwards, the test permanently reads (writes) consecutive disk blocks of a given size from (to) the disk, until a maximum size has been reached. We performed the test for block sizes from 0.5 KByte up to 32 KByte. We obtained the required energy per block size to run the benchmark from our accounting infrastructure.

The results for the read case are shown in Figure 7. The write case yields virtually the same energy distribution; for reasons of space, we do not show it here. For each size, the figure shows disk and CPU power consumption of the client and the device driver VM. The lowermost part of each bar shows the base CPU power consumption required by core components such as the hypervisor and the user-level VMM (36W); this part is consumed independently of any disk load. The upper parts of each bar show the active and idle power consumption caused by the stress test, broken into CPU and disk consumption. Since the client VM is the only consumer of the hard disk, it is accounted the complete idle disk power (3.5) and CPU power (8W) consumed by the driver VM. Since the benchmark saturates the disk, the active disk power consumption of the disk driver mostly stays at its maximum (2W), which is again accounted to the client VM as the only consumer. Active CPU power consumption in the driver VM heavily depends on the block size and ranges from 9W for small block sizes down to 1W for large ones. Note that the CPU costs for processing a virtual disk request may even surpass the costs for handling the request on the physical disk. Finally, active CPU power consumption in the client VM varies with the block sizes as well, but at a substantially lower level; the lower level comes unsurprising, as the benchmark bypasses most parts of the disk driver in the client OS. The thin bar on the right of each energy bar shows the real power consumption of the CPU and disk, measured with the external DAQ system.

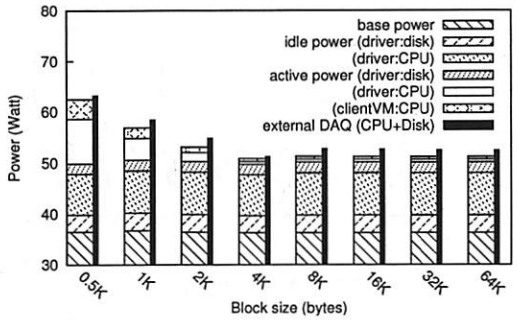


Figure 7: Energy distribution for CPU and disk during the disk stress test. The thin bar shows the real CPU and disk power consumption, measured with an external DAQ system.

4.2 Enforcing Power Constraints

To demonstrate the capabilities of VM-based energy allocation, and to evaluate the behavior of our disk throttling algorithm over time, we performed a second experiment with *two* clients that simultaneously

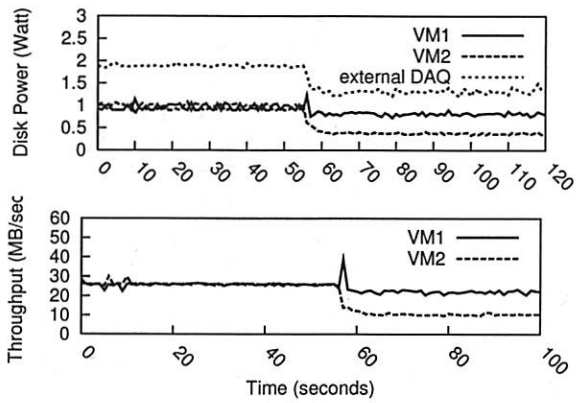


Figure 8: Disk power consumption and throughput of two constrained disk test simultaneously running in two different guest VMs.

require disk service from the driver. The clients interface with a single disk driver VM, but operate on distinct hard disk partitions. We set the active driver power limit of client VM 1 to 1W and the limit of client VM 2 to 0.5W, and periodically obtained driver energy and disk throughput over a period of about 2 minutes. Figure 8 shows both distributions; we set the limit about 45 seconds after having started the measurements. Our experiment demonstrates the driver's capabilities to VM-specific control over power consumption. The internal accounting and control furthermore corresponds with the external measurements.

4.2.1 Guest-Level Energy Allocation

In the next experiment, we compared the effects of enforcing power limits at the host-level against the effects of guest-level enforcement. In the first part of the experiment, we ran two instances of the compute-intensive bzip2 application within an energy-unaware guest OS. In the unconstrained case, a single bzip2 instance causes an active CPU power consumption of more than 50W. The guest, in turn, is alloted an overall CPU active power of only 40W. As the guest is not energy-aware, the limit is enforced by the host-level subsystem. In the second part, we used an energy-aware guest, which complies with the alloted power itself. It redistributes the budget among the two bzip2 instances using the resource container facility. Within the guest, we set the application-level power limits to 10W for the first, and to 30W for the second bzip2 instance. Note that the power limits are effective limits; strictly spoken, both bzip2 still consume each 50 Joules per second when running; however, the resource container implementation reduces the each task time accordingly, with the result that over time, the limits are effectively obeyed.

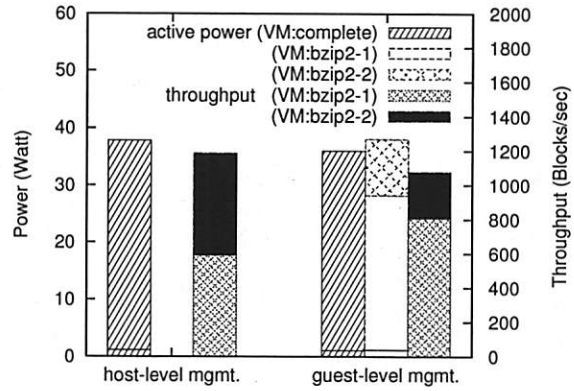


Figure 9: Guest-Level energy redistribution.

The results are given in Figure 9. For both cases, the figure shows overall active CPU power of the guest VM in the leftmost bar, and the throughput broken down to each bzip2 instance in the rightmost bar. For the energy-aware VM, we additionally obtained the power consumption per bzip2 instance as seen by the guest's energy management subsystem itself; it is drawn as the bar in the middle.

Note that the guest's view of the power consumption is slightly higher than the view of the host-level energy manager. Hence, the guest imposes somewhat harsher power limits, and causes the overall throughput of both bzip2 instances to drop compared to host-level control. We attribute the differences in estimation to the clock drift and rounding errors in the client.

However, the results are still as expected: host-level control enforces the budgets independent of the guest's particular capabilities – but the enforcement treats all guest's applications as equal and thus reduces the throughput of both bzip2 instances proportionally. In contrast, guest-level management allows the guest to respect its own user priorities and preferences: it allots a higher power budget to the first bzip2 instance, resulting in a higher throughput compared to the second instance.

5 Related Work

There has been a considerable research interest in involving operating systems and applications in the management of energy and power of a computer system [6, 7, 9, 10, 14, 25, 27]. Except for the approach of vertically structured OSes, which will be discussed here, none of them has addressed the prob-

lems that arise if the OS consists of several layers and is distributed across multiple components, as current virtualization environments do. To our knowledge, neither the popular Xen hypervisor [2,19] nor VMware's most recent hypervisor-based ESX Server [22] support distributed energy accounting or allocation across module boundaries or software layers.

Achieving accurate and easy accounting of energy by vertically structuring an OS was proposed by the designers of *Nemesis* [14,18]. Their approach is very similar to our work in that it addresses accountability issues within multi-layered OSes. A vertically structured system multiplexes all resources at a low level, and moves protocol stacks and most parts of device drivers into user-level libraries. As a result, shared services are abandoned, and the activities typically performed by the kernel are executed within each application itself. Thus, most resource and energy consumption can be accounted to individual applications, and there is no significant anonymous consumption anymore.

Our general observation is that hypervisor-based VM environments are structured similarly to some extent: a hypervisor also multiplexes the system resources at a low level, and lets each VM use its own protocol stack and services. Unfortunately, a big limitation of vertical structuring is that it is hard to achieve with I/O device drivers. As only one driver can use the device exclusively, all applications share a common driver provided by the low-level subsystem. To process I/O requests, a shared driver consumes CPU resources, which recent experiments demonstrate to be substantial in multi-layered systems that are used in practice [5]. In a completely vertically structured system, the processing costs and energy can not be accounted to the applications. In contrast, it was one of the key goals of our work to explicitly account the energy spent in service or driver components.

The *ECOSystem* [27] approach resembles our work in that it proposes to use energy as the base abstraction for power management and to treat it as a first-class OS resource. ECOSystem presents a *currentcy* model that allows to manage the energy consumption of all devices in a uniform way. Apart from its focus on a monolithic OS design, ECOSystem differs from our work in several further aspects. The main goal of ECOSystem is to control energy consumption of mobile systems, in order to extend their battery lifetime. To estimate the energy consumption of individual tasks, ECOSystem attributes power consumptions to different states of each device (e.g. standby, idle, and active states) and charges applications if they cause a device switch to a higher power state. ECOSystem does not distinguish between the fractions contributed by different devices; all cost that a task causes is accumulated to one value. This allows the OS to control the overall energy consumption without considering the currently installed devices. However, it renders the approach too inflexible for other energy management schemes such as thermal management, for which energy consumption must be managed individually per device.

In previous work [3], Bellosa et al. proposed to estimate the energy consumption of the CPU for the purpose of thermal management. The approach leverages the performance monitoring counters present in modern processors to accurately estimate the energy consumption caused by individual tasks. Like the ECOSystem approach, this work uses a monolithic operating system kernel. Also, the estimated energy consumption is just a means to the end of a specific management goal, i.e., thermal management. Based on the energy consumption and a thermal model, the kernel estimates the temperature of the CPU and throttles the execution of individual tasks according to their energy characteristics if the temperature reaches a predefined limit.

6 Conclusion

In this work, we have presented a novel framework for managing energy in multi-layered OS environments. Based on a unified energy model and mechanisms for energy-aware resource accounting and allocation, the framework provides an effective infrastructure to account, distribute, and control the power consumption at different software layers. In particular, the framework explicitly accounts the recursive energy consumption spent in the virtualization layer or subsequent driver components. Our prototypical implementation encompasses a host-level subsystem controlling global power constraints and, optionally, an energy-aware guest OS for local, application-specific power management. Experiments show that our prototype is capable of enforcing power limits for energy-aware and energy-unaware guest OSes.

We see our work as a support infrastructure to develop and evaluate power management strategies for VM-based systems. We consider three areas to be important and prevalent for future work: devices with multiple power states, processors with support for hardware-assisted virtualization, and multi-core architectures. There is no design limit with respect to the integration into our framework, and we are actively developing support for them.

Acknowledgements

We would like to thank Simon Kellner, Andreas Merkel, Raphael Neider, and the anonymous reviewers for their comments and helpful suggestions. This work was in part supported by the Intel Corporation.

References

[1] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 45–58, Berkeley, CA, Feb. 1999.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles*, pages 164–177, Bolton Landing, NY, Oct. 2003.

[3] F. Bellosa, A. Weissel, M. Waitz, and S. Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, pages 1–10, New Orleans, LA, Sept. 2003.

[4] R. Bianchini and R. Rajamony. Power and energy management for server systems. *IEEE Computer*, 37(11):68–74, 2004.

[5] L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference*, pages 387–390, Anaheim, CA, Apr. 2005.

[6] K. Flautner and T. N. Mudge. Vertigo: automatic performance-setting for Linux. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 105–116, Boston, MA, Dec. 2002.

[7] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th Symposium on Operating System Principles*, pages 48–63, Charleston, SC, Dec. 1999.

[8] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, Oct. 2004.

[9] M. Gomaa, M. D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 260–270, Boston, MA, Sept. 2004.

[10] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: dynamic speed control for power management in server class disks. In *Proceedings of the 30th annual international symposium on Computer architecture (ISCA)*, pages 169–181, New York, NY, June 2003.

[11] H. Härtig, M. Hohmuth, J. Liedtke, and S. Schönberg. The performance of μ-kernel based systems. In *Proceedings of the 16th Symposium on Operating System Principles*, pages 66–77, Saint Malo, France, Oct. 1997.

[12] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, and R. Bianchini. Mercury and Freon: temperature emulation and management in server systems. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 106–116, San Jose, CA, Oct. 2006.

[13] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 135–140, Huntington Beach, CA, Aug. 2001.

[14] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, Sept. 1996.

[15] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 17–30, San Fransisco, CA, Dec. 2004.

[16] Maxtor Corporation. *DiamondMax Plus 9 Data Sheet*, 2003.

[17] A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. In *Proceedings of the 1st EuroSys conference*, pages 403–414, Leuven, Belgium, Apr. 2006.

[18] R. Neugebauer and D. McAuley. Energy is just another resource: Energy accounting and energy pricing in the nemesis OS. In *Proceedings of 8th Workshop on Hot Topics in Operating Systems*, pages 67–74, Schloß Elmau, Oberbayern, Germany, May 2001.

[19] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Malick. Xen 3.0 and the art of virtualization. In *Proceedings of the 2005 Ottawa Linux Symposium*, pages 65–85, Ottawa, Canada, July 2005.

[20] J. Stoess and V. Uhlig. Flexible, low-overhead event logging to support resource scheduling. In *Proceedings of the Twelfth International Conference on Parallel and Distributed Systems*, volume 2, pages 115–120, Minneapolis, MN, July 2006.

[21] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, San Jose, CA, May 2004.

[22] VMware Inc. *ESX Server Data Sheet*, 2006.

[23] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 1–11, Monterey, CA, Nov. 1994.

[24] A. Weissel and F. Bellosa. Dynamic thermal management for distributed systems. In *Proceedings of the 1st Workshop on Temperature-Aware Computer Systems*, Munich, Germany, May 2004.

[25] A. Weissel, B. Beutel, and F. Bellosa. Cooperative IO - a novel IO semantics for energy-aware applications. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 117–130, Boston, MA, Dec. 2002.

[26] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang. Modeling hard-disk power consumption. In *Proceedings of the Second Conference on File and Storage Technologies*, pages 217–230, San Francisco, CA, Mar. 2003.

[27] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: managing energy as a first class operating system resource. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, San Jose, CA, Oct. 2002.

[28] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Currentcy: unifying policies for resource management. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 43–56, San Antonio, TX, June 2003.

Xenprobes, A Lightweight User-space Probing Framework for Xen Virtual Machine

Nguyen Anh Quynh, Kuniyasu Suzaki
National Institute of Advanced Industrial Science and Technology, Japan.
{nguyen.anhquynh,k.suzaki}@aist.go.jp

Abstract

This paper presents *Xenprobes*, a lightweight framework to probe the guest kernels of Xen Virtual Machine. Xenprobes is useful for various purposes such as as monitoring real-time status of production systems, analyzing performance bottlenecks, logging specific events or tracing problems of Xen-based guest kernel. Compared to other kernel probe solutions, Xenprobes introduces some unique advantages. To name a few: First, our framework puts the the breakpoint handlers in user-space, so it is significantly easier to develop and debug. Second, Xenprobes allows to probe multiple guests at the same time. Last but not least, Xenprobes supports all kind of Operating Systems supported by Xen.

1 Introduction

Testing and debugging Operating System (OS) kernel is a hard and tired job of kernel developers. An easy and intuitive solution like inserting debug code (such as *printk()* function in Linux) into the kernel source, then recompile it and reboot the system is widely used. Nevertheless, this technique has a major drawbacks: It is very time-consuming and slow, especially because compiling kernel might take no less than 30 minutes on old machines. Moreover, in general commercial OS-es do not provide the source code for us to modify and recompile in the first place.

One solution to the problem is to use kernel debugger [16] [10]. However, while kernel debuggers allow developers to inspect kernel at run-time, a debugger is not always desirable because it requires user interactivity. In case the testing and debugging must be done automatically, for example to monitor system status in a long time, this approach is not up to the task.

Hence the advent of dynamical *probing* technique, which allows the developers - at run-time - to specify the actions when corresponding events happen in the kernel. Basically this technology dynanmically inserts a breakpoint into a running kernel, without having to modify the kernel source. The place of the breakpoint, usually specified by its address in kernel address space, is called *probe-point*. Each *probe* associates a probe-point with a *probe handler*. A probe handler runs as extension to the system breakpoint interrupt handler and usually has little or no dependence on system facilities. Probes are able to be inserted in the almost everywhere in the kernel, like interrupt-time, task-time, disabled, inter-context switch and SMP-enabled code paths, etc...

In Linux world, such a probe framework is *Kprobes*. Kprobes was merged into Linux kernel from version 2.6.9, and provides a lightweight interface for kernel modules to inject probes and register corresponding probe handlers. Kprobes are intended to be used in test and development environments. During test, faults may be injected or simulated by the probing module. In development, debugging code, like a *printk()* function, may be easily inserted without having to recompile the kernel.

However, solutions like Kprobes have some major shortcomings. Here are the most notable drawbacks:

1. Probe handlers come in the shape of kernel code, specifically in kernel module. The problem is that comparing to programming in user-space, programming in kernel context is limited by many restrictions, such as allocating and accessing resource, short on available library, no floating-point math, etc... Programming for OS kernel is considered very complicated and tricky, thus usually requires highly experienced developers, because a broken code can make the whole system in-stable.

   That is a reason why programming in user-space is always encouraged over kernel-space. In fact, there is a basic principle within kernel developer community: Whatever can be done in user-space, never do it in kernel-space unless absolutely necessarily. Various attempts from different open source

projects trying to submit their work to Linux kernel are rejected, and asked to re-architecture their code to work in user-space instead.

2. Kprobes cannot put the probes in some places in the kernel, such as in the code that implements Kprobes and functions like *do_page_fault()* and *notifier_call_chain()* [8].

3. Kprobes makes no attempt to prevent probe handlers from stepping on each other – for example it is not a good idea to probe *printk()* and then call *printk()* from inside the probe handler. If a probe handler hits a probe, then the second probe's handlers will not run in that instance, and the *kprobe.nmissed* member of the second probe will be incremented [8].

4. Kprobes is designed and works for Linux only, thus the probing code made for Kprobes cannot be easily reused for other OS-es.

Recently virtual machine (VM) technology has emerged as one of the hottest topics in computer research. The principle of VM technology is to allow the creation of many virtual hosts running at the same time on the same physical machine, each running an instance of an OS. Obviously VM software such as Xen Virtual Machine [5] [14] can help to reduce both hardware and maintenance costs for organizations that need to use various machines for different services. Especially Xen even offers a convenient method of debugging OS kernel, so it is possible to debug an OS like debugging an user-process [1].

Taking the advantage of Xen technology, this paper proposes a framework named *Xenprobes* for probing Xen-based guest OS kernel. Similar to Kprobes, Xenprobes allows developers to dynamically inject probes into guest VMs at run-time. However, Xenprobes is able to address the above-mentioned problems of Kprobes:

1. Xenprobes' handlers completely work in user-space, therefore significantly easier than Kprobes to develop the handlers, even with less experienced programmers. This includes the benefit of using any library available in user-space.

2. Xenprobes can put the probes at any place in probed VMs without worrying about conflict.

3. Because the Xenprobes handlers run in user-space instead of in the kernel of the probed VM, we eliminate the problem of stepping on other handlers like in Kprobes' case.

4. Xenprobes is OS-independent, and can provide service for any OS supported by Xen. In addition, Xenprobes is designed to support probing multiple VMs at the same time.

The rest of this paper is organized as followings. Section 2 briefly covers some background of Xen Virtual Machine and Xen debugging technique. Section 3 presents the architecture and implementation of Xenprobes framework, while section 4 discusses some issues that can be raised when using the framework. Section 5 evaluates the performance overhead of our framework in Linux guest kernels. Section 6 summaries the related works, and compare their advantage as well as drawbacks with our approach. Finally we conclude the paper in section 7.

2 Background on Xen Virtual Machine

Our framework Xenprobes is based on Xen, and exploits the debugging architecture of Xen to inject software breakpoints into probed VM. In this part, we will take a brief look at Xen technology. After that we discuss the kernel debugging architecture for Xen VM version 3.0.3, the latest version as of this writing, which is used by our Xenprobes.

2.1 Xen Virtual Machine

Xen is an open source virtual machine monitor initially developed by the University of Cambridge Computer Laboratory and now promoted by various industrial players like Intel, AMD, IBM, HP, RedHat, Novel and by the open source community. Xen can be used to partition a machine to support the concurrent execution of multiple operating systems (OS). Xen is outstanding because the performance overhead introduced by virtualization is negligible: the slowdown is around 3% only ([4]). Various practices take the advantages offered by Xen, such as server consolidation, co-located hosting facilities, distributed services and application mobility, as well as testing and debugging software.

Basically, Xen is a thin layer of software running on top the bare hardware. This layer is either called hypervisor or virtual machine monitor. Its main job is to provide a virtual machine abstraction to the above OS-es. Running on top of Xen, VM is called Xen domain, or domain in short. A privileged special domain named *Domain0* (or *Dom0* in short) always runs. Dom0 controls other domains (called *User Domain*, or *DomU* in short), including jobs like start, shutdown, reboot, save, restore and migrate them between physical machines. Especially, Dom0 is able to map and access to memory of other DomUs at run-time.

Initially, Xen only supports the *para-virtualization* technique, in which Xen exposed a hardware architecture called *xen* to the VMs, and the OS-es running on Xen must be modified to work with *xen*.

Recently, realizing the potential of virtualization, Intel and AMD launch special CPUs that support virtualization at lowest level [7] [2]. Xen takes the advantage of these processors to provide the *full-virtualization* technique, in which all OS-es can run on Xen without any modification.

2.2 Exceptions Handling in Xen

Xen handles exceptions differently with para-virtualization and full-virtualization.

- **Para-virtualization**: In Xen, to manage VMs and the physical hardware, the hypervisor layer runs at the highest privilege level (ring 0 in the case of x86 architecture). To provide a strong isolation between VMs as well as between VMs and the hypervisor, all the VMs are modified run at lower level (ring 1 in the case of x86 architecture). So are the interrupt handlers of VMs: While normally the interrupt handlers are registered in the interrupt descriptor table (IDT), Xen does not allow VMs to install their handlers themselves because of the security reasons: it cannot give VMs the direct access to the below hardware. Instead, VM kernels are modified at source code, so the hypervisor captures the interrupts instead of letting the VMs handle them.

  Specifically, in the asynchronous interrupt case, also called exception and generated when the system executes the *INTO*, *INT1*, *INT3*, *BOUND* instructions or caused by page faults: these exceptions are processed in the hypervisor layer first instead in the VM's kernel. To register handlers, a VM's kernel is modified to call the hypercall named *HYPERVISOR_set_trap_table* to setup the exception handlers. The handlers are functions initialized at machine boot time, and managed by hypervisor layer.

- **Full-virtualization**: Virtualization-enable processors such as Intel-VT and AMD-V support full virtualization by adding new privileged ring, which is at a higher privilege level than ring 0. Xen runs this privilege, and lets the OS-es run in ring 0. In ring 0, the OS-es run as normally without being aware that it is managed by the hypervisor run below. Xen virtualizes the processor for VMs on it by intercepting special instructions and exceptions. In case of debugging related instructions *INT1* and *INT3*, these interrupts are intercepted as privilege ring transitions, then virtualized exceptions are injected into related VMs.

  For more detail, readers are encouraged to read the paper [1].

2.3 Debugging Support Architecture in Xen

Similarly to exceptions, there is a little difference in the way Xen supports debugging in para-virtualization and full-virtualization.

- **Para-virtualization**: In x86 architecture, *INT3* is a breakpoint instruction which is used for debugging purpose [1]. Whenever this instruction is hit, the control is passed to the exception handler of *INT3* in kernel space. In Xen, the sequence of handling the *INT3* exceptions is as in the following steps:

  - When the VM hits the breakpoint instruction, the exception *#BP* is raised.
  - The system makes a hypervisor switch to give control to the *INT3* handler staying in the hypervisor layer.
  - The *INT3* handler in hypervisor checks if VM is in kernel mode. If that is not the case, Xen returns the control to VM.
  - If the exception comes from VM's kernel, Xen pauses the VM for inspection.

  In fact, the Xen debugger works by exploiting the mentioned feature: When the debugger server running in Dom0 detects that the concerned domain is paused, it comes to inspect the VM's kernel, then resume it after it finishes the job [1].

  Besides *INT3*, *INT1* is another special interrupt made for debugging. This interrupt sends the processor into the single-step mode, in which after each construction, the handler of *INT1* is called. To make this happen, we only need to enable the trap flag *TF* of the *FLAGS* register. The processor switches to normal mode if the *TF* flag is turned off. And similarly to the case of *INT3*, when the system is in single-step mode, after each instruction the control is changed to the INT1 handler at hypervisor layer. The sequence of handling *INT1* is same as in *INT3*'s case.

- **Full-virtualization**: The way Xen handles *INT1* and *INT3* in full-virtualization is quite straightforward: when the processor hits *INT3*, a *#BP* exception is raised, resulting in a privilege transition from to the hypervisor. The hypervisor then simply pauses the VM for inspection.

[1] Breakpoint instruction is an one-byte opcode with the value of *0xCC* on x86 platform.

The processing of *INT1* is similar. For more detail, please see [1].

3 Xenprobes Architecture and Implementation

Xenprobes works by modifying the kernel code of probed VMs at run-time, in which it replaces the instruction at the probe-point with a breakpoint instruction. Xenprobes provides a very lightweight and simple framework, so the developers can easily employ it and writes his probing code in user-space of Dom0.

In this section, we present the framework, then go into detail of the implementation of Xenprobes.

3.1 Xenprobes Framework

In the design of Xenprobes, we take advantage of the Xen debugging infrastructure: We notice that if we put a breakpoint into a domain memory, whenever the breakpoint is hit in execution, the control is given to the hypervisor, and the domain is paused for inspection. So if we put the breakpoint handlers in user-space of Dom0 and somehow inform Dom0 about the breakpoint event, Dom0 can run the corresponding handler and let the handler does everything in user-space.

3.1.1 Xenprobes Types

Xenprobes allows the developers to dynamically inject breakpoints into any place in a Xen VM, and collect debugging and performance information non-disruptively. We can trap at any kernel code address, specifying handler functions to be invoked when the probe-point is hit.

We define a Xen probe (or probe in short from now on) as a set of handlers placed on a certain instruction address of a specific VM. For convenience, the instruction address is set in *virtual address* of that VM, so the developer can take the address from the kernel symbol file accompanied the VM kernel binary.

Currently, Xenprobes supports two types of probes, *XProbe* - or *XP* in short - and *XrProbe* - or *XrP* in short. An XP can be placed on any instruction in the kernel. An XrP is put in the entry of a specified function, and fired when entering and leaving the function. XrP can be used to collect information such as parameters and returned value of kernel functions.

Each XP comes with a pair of functions called *pre-handler* and *post-handler*. When a breakpoint of a particular VM is hit, the corresponding pre-handler is executed just before the execution of the probed instruction. The post-handler is executed right after the execution of the probed instruction.

An XrP is also accompanied by two handlers: a *entry-handler* and a *return-handler*. The entry-handler is executed when the probe is first hit, usually when the execution enters the probed function. This is the approriate time to gather the parameters of function. When the probed function returns, the return-handler is fired, and the returned value of the function can be collected.

The handlers can do other things such as check and modify registers, inspect and change the memory content of the probed VM. In addition, because Xenprobes handlers operating in user-space context of Dom0, it is possible to use use any library available in user-space, as with any other user-space application. In contrary, Kprobes handlers run in kernel context instead, thus cannot have such flexibility.

3.1.2 Xenprobes Framework

The Xenprobes framework has been designed in such a way that tools for debugging, tracing and logging could be built by extending it. The framework provides an user-space library, so the developers can use it to write their probing applications.

In the design, Xenprobes supports multiple hardware architectures and hides all the internal complexity in order to give developers a very simple interface to work with. The framework's interface is described in C programming language with some newly-defined data types and seven functions as in Figure 1. To employ the framework, developers simply include a C prototype header file named *xenprobes.h*, and compile their code with *xenprobes* library.

Below is a brief summary of Xenprobes interface.

- *xenprobes_handle_t*: Xenprobes defines a new data type to manage probes, named *xenprobes_handle_t*. Both XP and XrP can be referred to using a variable of this data type.

- *xenprobes_handler_t*: The breakpoint handler can be defined using a new function pointer type *xenprobes_handler_t*. The handler is a function with two arguments: a probe handle of *xenprobes_handle_t* type and a pointer to the virtual cpu structure of *cpu_user_regs*. [2]

- *register_xenprobe()*: An XP must be registered using this function. *register_xenprobe()* requires four arguments: The first is a domain id, which indicates which Xen VM we want to probe. The second argument is the address where we want to inject the breakpoint. For convenience, this is the virtual address in the VM. The third and fourth argument are

[2] *cpu_user_regs* is a data type defined by Xen and can be referred to by including a C header file *xenctrl.h* from Xen's *libxc* library.

pre-handler and post-handler, which are called before and after the original instruction at the probepoint is executed. One and only one of these last two arguments can be *NULL*, and can be used if we want to ignore the pre- or post- event.

This function returns a handle which will be used as the first argument of the probe handlers when they are called. *XENPROBES_HANDLE_ERROR*[3] value is returned in case there is a problem when registering the breakpoint.

- *unregister_xenprobe()*: An XP can be unregistered using this function. *unregister_xenprobe()* takes only one parameter, which is a handle returned when that XP was registered.

- *register_xenretprobe()*: An XrP must be registered using this function. *register_xenretprobe()* is quite similar to the *register_xenprobe()*, but the address argument specifies the entry address of the function we want to probe. The *entry_handler* argument specifies the entry-handler function executed right before the breakpoint instruction is hit, and the *return_entry* argument specifies the return-handler function executed right before the probed function returns. The last argument *maxactive* indicates how many instances of the specified function can be probed simultaneously.

  Without the XrP, if we need to inspect the return point of a function, we may have to put multiple XPs to cover multiple code paths. But with we use XrP, we only need to put a single probe at the entry of the function, and it automatically fires whenever that function returns, regardless of how it exits.

- *unregister_xenretprobe()*: Similar to XP's case, an XrP must be unregistered using this function, which takes an XrP handle as the only argument.

- *xenprobes_loop()*: When all the probes are successfully registered, we can start probing the VMs with this function. *xenprobes_loop()* sends us into an infinite loop, in which Xenprobes waits for the debugging events, that indicates that a particular VM is waiting for probing, and executes the corresponding handlers. This infinite loop only quits if the *xenprobes_stop()* function below is called.

- *xenprobes_enable()*: The probe handlers can call *xenprobes_enable()* function to turn on or turn off another probe. This function takes two no arguments: the first is corresponding handle, and the second argument indicates if we want to enable (if *active* is *1*) or disable (if *active* is *0*) the probe.

- *xenprobes_stop()*: The probe handlers can call this function to stop probing. *xenprobes_stop()* takes no argument, and will immediately get *xenprobes_loop()* out of its loop.

[3] *XENPROBES_HANDLE_ERROR* is a constant value defined in *xenprobes.h*

```
typedef unsigned long xenprobes_handle_t;

typedef int (*xenprobe_handler_t)(
    xenprobes_handle_t,
    struct cpu_user_regs *);

xenprobes_handle_t register_xenprobe(
    domid_t domid,
    unsigned long address,
    xenprobe_handler_t pre_handler,
    xenprobe_handler_t post_handler);

int unregister_xenprobe(
    xenprobes_handle_t handle);

xenprobes_handle_t register_xenretprobe(
    domid_t domid,
    unsigned long address,
    xenprobe_handler_t entry_handler,
    xenprobe_handler_t returned_handler,
    int maxactive);

int unregister_xenretprobe(
    xenprobes_handle_t handle);

int xenprobes_loop(void);

int xenprobes_enable(
        xenprobes_handle_t handle,
        int active);

void xenprobes_stop(void);
```

Figure 1: Xenprobes framework.

We demonstrate the usage of our framework in two simple examples in the Figure 2 and Figure 3. Figure 2 gives some hints on how to use an XP to monitor a Linux VM running on Xen. The XP is registered to watch the *sys_open()* function, so it can notify us each time the *open* system-call is executed. For brevity, the sample assumes that the probed VM has domain id of *1*. We get the virtual addresses of the *sys_open()* function, which is available with the *sys_open* symbol at the address *0xc01511d0* from the symbol file *System.map* accompanying the binary kernel of the Linux VM. The pre-handler of the XP,

function *xp_open()*, prints out each time the *open* system-call is executed, for example when we *read* a file, and quits the loop after *10* times by calling *xenprobes_stop()* function. Finally, it removes the XP after the loop with *unregister_xenprobe()*.

Sample in figure 2 hints us how to probe with an XrP. The XrP is registered to watch the *sys_unlink()* function (at the address *0xc0161780*, corresponding to symbol *sys_unlink* in *System.map* file.) of domain *1*, so it can notify us the function parameters and returned value each time the *unlink* system-call is executed. The entry-handler *xrp_entry_unlink()* prints out the address of the *path-name* parameter of the system-call, which is retrieved from the second integer above the stack pointer *ESP*. The return-handler *xrp_return_unlink* prints out the returned value of the system-call, retrieved from the *EAX* register. This returned value indicates the result when when a file is *removed*. We want to probe the *unlink* system-call at most *8* times at a time. The probe also quits the loop after *5* times by calling *xenprobes_stop()* function. Finally, it unregisters the XrP after the loop.

These samples must be compiled with the Xenprobes library and run in Dom0.

```
...
static int xp_open(
    xenprobes_handle_t handle,
    struct cpu_user_regs *regs)
{
   static int count=0;
   count++;
   printf("sys_open: %d\n", count);
   if (count == 10)
       /*quit probe looping*/
       xenprobes_stop();
   return 0;
}
...
xenprobes_handle_t h;

h = register_xenprobe(
   1,          /*domain id*/
   0xc01511d0,/*sys_open address*/
   xp_open,   /*XP handler*/
   NULL);     /*No post-handler*/

xenprobes_loop(); /*Enter the loop*/
unregister_xenprobe(h);
...
```

Figure 2: A simple example on how to use XP.

```
...
static int xrp_entry_unlink(
    xenprobes_handle_t handle,
    struct cpu_user_regs *regs)
{
   unsigned long *stack;
   static int count=0;
   count++;
   stack = &regs->esp;
   /*stack[0] = returned address*/
   printf("sys_unlink: path-name @%x\n",
          (unsigned int)stack[1]);
   if (count == 5)
       /*quit probe looping*/
       xenprobes_stop();
   return 0;
}

static int xrp_return_unlink(
    xenprobes_handle_t handle,
    struct cpu_user_regs *regs)
{
   static int count=0;
   count++;
   /*get the returned value in EAX*/
   printf("sys_unlink returned: %d\n",
          regs->eax);
   if (count == 5)
       /*quit probe looping*/
       xenprobes_stop();
   return 0;
}
...
xenprobes_handle_t h;

h = register_xenretprobe(
   1,          /*domain id*/
   0xc0161780, /*sys_unlink addr*/
   xrp_entry_unlink, /*entry-handler*/
   xrp_return_unlink,/*return-handler*/
   8);  /*At most 8 probes handled*/

xenprobes_loop();  /* Enter the loop */
unregister_xenretprobe(h);
...
```

Figure 3: A simple example on how to use XrP to retrieve function parameters and returned value.

3.2 Xenprobes Implementation

Xenprobes heavily depends on specific features of processor architecture and uses different mechanisms depending on the architecture on which it is being executed. At the moment, Xenprobes is available on i386 and x86_64 architectures.

Xenprobes is provided in a shaped of an user-space library named *xenprobes* in Dom0. Totally the code is around less than 4000 lines of C source code, in which the architecture-dependent code is around 1000 lines.

3.2.1 Performance Challenges

One of the first challenges when we implemented Xenprobes is the performance penalty problem: every time a breakpoint is hit leads to several hyper-switches: first is a switch from the probed VM kernel to the hypervisor; second is a switch from hypervisor to Dom0 to have Xenprobes handled the breakpoint event; and finally the control is given back to the probed VM. These switchings can cause a lot of negative impact to to the overall performance of the probed VM.

When we first investigated the problem, we thought it was a good idea to employ the same tactic of the current Xen kernel debugger to monitor VM, because the Xen debugger also exploits breakpoint mechanism to inspect VM's kernel at run-time ([1]). However, this approach has a disadvantage that can badly affect the system performance: the debugger in Dom0 detects the debugging event by periodically polling VM's status to see if it is paused [4], which is the evidence that the breakpoint was hit. By default the checking interval is 10 million nanoseconds, which means breakpoints cannot be processed immediately if they come between the checking time. For debugging purpose, that is not a major concern because performance is not a priority. But for our target, that is unfortunately unacceptable because the whole process slows down significantly.

To address the problem, we decide not to adopt the mentioned polling tactic of the Xen kernel debugger. Instead we exploit a special feature of the debugging architecture in Xen: no matter whether the VM is paravirtualization or full-virtualization, whenever the hypervisor gets debugging control given to it from its VMs [5], the hypervisor sends an event to Dom0 to notify any potential debugger running there. While the standard debugger does not use this feature, we do employ it, and have Xenprobes handled the debugging event. To do that, Xenprobes only needs to put the protected VM into the debugging mode [6], and binds to the virtual interrupt *VIRQ_DEBUGGER*, which is dedicated for debugging event, to get notified by the hypervisor. Thanks to this strategy, Xenprobes is instantly aware when probed VMs hits the breakpoints, therefore does not need to poll VMs for the paused status. Some experiements demonstrate that our approach significantly improves the overall performance.

3.2.2 Access VM's Kernel Memory

In Xenprobes architecture, we need to read and write to VM's kernel memory, for example to read the original instruction at probe-point and overwrite it with the breakpoint. In order to access to a specific virtual address of VM, we must first translate it into physical address. Currently Xen support several kinds of architecture: *x86_32*, *x86_32p* and *x86_64*, and each of these platforms has different schemes of paging memory. Hence Xenprobes must detect the underlying hardware, and then translates the virtual memory accordingly by traversing the page table tree.

To traverse the page table tree, it is imperative to know the physical address of the page directory. In Xen, we can have the virtual control register *cr3* of each virtual CPU of VM by getting corresponding CPU context via Xen function *xc_vcpu_getcontext()* [17]. Besides, as Xen supports several architectures such as *x86*, *PAE* and *x86_64* (thus different page-table formats), Xenprobes must handle the page-table accordingly to convert the virtual address to physical address.

After that, Xenprobes accesses memory of VM by mapping the physical address with the function named *xc_map_foreign_range()* [17]. Then it goes on reading or writing to the mapped memory [7].

To ensure integrity, each read or write access to the VM requires pausing it, and we need to resume it back after finishing.

3.2.3 Out-of-line Execution Area

Regarding the technique of handling the breakpoints, Xenprobes adopts the same solution proposed by Kprobes [8] [11], with some modifications. Basically Xenprobes replaces the instruction at the probe-point with a breakpoint. The original instruction at that point is copied to a separate area, which is executed when Xenprobes handles the breakpoint event. We call this area "*Out-of-line Execution Area*" or *OEA* in short.

Regarding the size of each OEA: besides storing the original instruction, each OEA must also have enough

[4] This can be done thanks to the Xen's *libxc* function *xc_waitdomain()*.

[5] This means either the processor hits the breakpoint, or it is put into the single-step mode

[6] This can be done with a domain control hypercall, with the special command *XEN_DOMCTL_setdebugging*.

[7] This depends on the mapped access is *PROT_READ* (read) or *PROT_WRITE* (write).

space for one relative jump instruction, which is used to jump back to the instruction next to the original instruction. Because the instruction size varies on different architectures, we move the code handling OEA into the architecture layer of Xenprobes.

This approach raises a question: how to have the OEA for each probe? Kprobes solves this issue simply by allocating an area of memory as OEA for each registered probe, and frees the OEA when unregistering the probe. In principle, a new OEA is assigned on demand whenever there is a need for a new probe.

However, this technique cannot be employed in our case: the probes are registered from user-space of Dom0 instead of from inside the corresponding kernel as with Kprobes. If we need to allocate an OEA for a new probe, there is no clean way to ask the probed VM to allocate the new chunk of memory inside its kernel for us.

We solve the problem by a simple method: We pre-allocate a fixed area of memory from inside the probed VM to store the OEAs for upcoming probes. We split the area into contiguous, non-overlap chunks of memory, and each chunk can be used as one OEA. When a new probe is registered, a free chunk will be given to that probe and the probe will use it as OEA. Xenprobes manages all the chunks from Dom0 with a bitmap structure, which indicates which chunk is in-use, which chunk is still free, thus can be allocated. Whenever a probe is unregistered, its associated OEA chunk is recovered for other demands.

To allocate the area of memory for OEAs, each VM that wants to support Xenprobes must be loaded with a kernel module. We provides such a module for Linux VM, named *xenprobesU*. This module is very simple: the only job of it is to allocate a configurable size of memory. The virtual address of this area and its size are then sent to Dom0 using the *XenBus* [17] interface . These values will be picked up from the *Xenstore* [17] in Dom0, and Xenprobes can then determine how many OEAs are available for each probed VM. More discussions on this issue are delayed to section 4.

3.2.4 Probes Registration

There are some differences on the way Xenprobes processes registration for XP and XrP.

- **XP Registration**: Registering an XP probe leads to allocate a dynamic memory in user-space of Dom0 for a new probe. All the probes are managed in a hash list, and the new probe is added to the list. Similarly to Kprobes, Xenprobes supports multiple probes, called *aggregate probes* by Kprobes, at the same probe-point. That allows the developers to register more than one probe at the same instruction address. All the probes have a list of aggregate probes named *aggregate list*, which is *NULL* normally. When a new probe needs to register at the same address, Xenprobes puts it in the *aggregate list* of the first probe, and execute the handlers of all the probes in the list, one by one and in the order, when the breakpoint at that address is hit at execution.

  After allocating memory for the new probe, the control is given to the architecture dependent code, in which the probe is prepared according to specific characteristics of the architecture. Xenprobes gets one free OEA for the probe. Then the instruction at the probe-point is copied to the OEA. Note that OEA stays inside the probed VM, so this steps requires one read and one write access to the VM's memory: the instruction is firstly read from the probe-point, then it is immediately written to the OEA.

  To speed up the procedure of breakpoint handling, we employ the *booster* technique proposed by Kprobes started from Linux kernel 2.6.17 [1]. The instruction at the probe-point is first checked to see if it is *boostable* [8]. The boostable instruction makes the probe boostable, and allow us to execute the instruction as if in inline case. The trick is to use a relative jump instruction to come back to the instruction next to the probe-point. This technique allows us to skip the single-step mode, thus significantly improve the performance.

  Any failure in allocating memory or preparing the probe return the error *XENPROBES_HANDLE_ERROR*. The system variable *errno* will be set to indicate what went wrong.

  An XP probe can be unregistered after finishing probing. The framework function *unregister_xenprobe()* removes the probe from the hash list of probes, then frees the OEA for later usage, and recovers the original instruction at the probe-point. After that, execution hit this point will not raise Xenprobes handler any more.

- **XrP Registration**: XrP actually builds on top XP to avoid duplicating code. When registering an XrP with the *register_xenretprobe()* function, Xenprobes puts a *entry-XP* at the entry of the probed function, and uses the entry-handler argument as the pre-handler of the entry-XP. Note that this entry-XP does not have the post-handler (in fact its post-handler has the *NULL* value). When the probed

[8] Boostable instructions are all instructions not belong to the set of *unboostable* instructions like relative jumps, relative calls or instructions that has hardware side-effect [11].

function is executed and this entry-XP is hit, Xenprobes saves a copy of the function's return address, and replaces it with the address of a *trampoline-XrP* in the probed VM.

The trampoline-XrP routine is actually a piece of code provided by the *xenprobesU* module, in which its only job is to execute a breakpoint (*INT3* instruction in i386 case). The address of this trampoline-XrP is also sent to Xenprobes via XenBus/Xenstore at initialization time together with the information about OEA memory.

The *maxactive* parameter specifies how many instances of the function can be probed at the same time, and *register_xenretprobe()* will preallocate enough memory to save the return address of the function. For example, if the function is non-recursive and is called with a spinlock held in the kernel, *maxactive* can get the value of 1. If the function is non-recursive and can never relinquish the CPU (like via a semaphore or preemption), we can set this parameter to the number of virtual cpus that the probed VM has.

One problem Xenprobes must handle is that when a VM is shutdown, all the probes as well as OEA memory is gone. Regarding this issue, Xenprobes watches for the shutdown VM by registering the built-in Xen watch *@releaseDomain* [17]. When this watch is fired, which indicates that the VM is not existent anymore, Xenprobes removes all the probes of the related VM.

3.2.5 Handling Probes

After probe registration step, the probing process starts by executing *xenprobes_loop()*. Firstly, this function registers to get notified by the hypervisor on debugging event by binding to the virtual interrupt *VIRQ_DEBUGGER*. Then it switches all the probed VMs to debugging mode and enters an infinite loop. In this loop, we listens for the debugging events sent from Xen. When a breakpoint is hit inside the kernel of a probed VM, the hypervisor pauses it, and sends a debugging event to notify Dom0. As the xenprobes_loop() registers to get the event, it comes up to handle the breakpoint. The procedure at this step is also different for XP and XrP, as followings.

- **XP Handling**: The XP is processed in the following sequences:

  (1) Find the XP probe related to this debugging event. After that we get the registers of the probed VM [9], which is later given to the probe handlers as a function parameter. If the processor is handling single-step mode, go to the step (3). Otherwise, we are handling breakpoint, so we execute the *pre-handler* of the probe. In case this is an aggregate probe, execute all the pre-handlers got from the aggregate list.

  (2) Point the instruction register [10] of the probed VM to the head of the OEA of the probe. As the OEA saves the original instruction at the probe-point, this will execute the original instruction. Then if the instruction is boostable, and there is no post-handler, go to the step (4). Otherwise, put the probed VM into the single-step mode [11]. The modified VM context which includes the new register value is updated [12]. After that, the probed VM is resumed.

  (3) The instruction in the OEA is executed. As the processor is in single-step mode, it traps back to the hypervisor, and another debugging event is sent to *xenprobes_loop()*. At this point, we execute the *post-handler* of the probe. After that, we fix the instruction pointer to point it to the instruction next to the original instruction at the probe-point. Then we resume the probed VM, and let it run normally, and wait for the the next breakpoint event (Do not go to the next step, as it handles the boostable-only instruction).

  (4) The instruction is boostable, so we just run the original instruction by pointing the instruction pointer to OEA, update the VM's context and resume the probed VM. The loop repeats with the next debugging events.

Actually there is a hidden procedure in the step (3) above: We must prepare for the boost in the first ever single-step time in case the instruction is boostable. We use information on the first probe hit to determine the location to put the jump instruction after the instruction at the head of OEA, as presented in [1]. So even if the instruction is boostable, we will execute it in single-step once in the first ever time the breakpoint is hit, but from the second time we can skip it and just run the OEA directly.

In conclusion, an XP causes at least one hyperswitch to Dom0. Ideally, probed instruction is boostable and there is no post-handler, Xenprobes

[9] Registers of a VM can be retrieved with the *libxc* function xc_vcpu_getcontext().

[10] On i386, that is the EIP register.

[11] On i386, this can be done by enabling the trap flag (TF) of the FLAGS register.

[12] Xen VM context can be set with the *libxc* function *xc_vcpu_setcontext()*.

only has to switch out once to execute the pre-hancler in Dom0. In the worse case, when the above condition is not satisfied, we have to single-step the probed instruction in OEA, thus need the second switch after single-step. Of course in term of performance, that causes negative impact to the VM.

- **XrP Handling**: Note that because XrP employs XP in its design, the handling procedure of its XP is similar to the above descriptions. However, there are few differences when it comes to execute its handlers.

  - When the probed function is executed, its entry-XP is hit. We switch out to Dom0 and execute its pre-handler, which is in fact the entry-handler of the registered XrP. After that, Xenprobes saves the return address of the probed function in a structure for the corresponding XrP. Remember that when registering the XrP, we must specify how many instances of the function can be probed simultaneously with the argument *maxactive*. However, if this limit is reached, Xenprobes does not save the return address, but simply increases the missed number in the XrP structure for the developers to investigate.

    After the above steps, Xenprobes overwrites the return address of the probed function with the address of the trampoline-XrP, and resumes the VM.

  - When the probed function returns, the trampoline-XrP is called. This code executes a breakpoint instruction as explained above. This action causes another switch to Dom0, and lets Xenprobes execute the return-handler of the XrP. After that, Xenprobes overwrites the return address of the probed function, and point the instruction pointer to this address. Finally, Xenprobes resumes the VM, which continues to execute at the return address of the probed function. The loop repeats and Xenprobes continues to wait for the next probes.

In conclusion, an XrP causes at least two hyperswitches to Dom0. In case the entry instruction is boostable (which is mostly the case), we have the first switch when the probed function is hit at entry time. At the end, when the function returns we have to switch out once more to execute the return-handler.

In the worse case, when the entry instruction is not boostable, we have to suffer two more hyperswitch for single-step: one is for the entry probe, another is for the return probe. However, as the first few instructions of the function prologue never have *unboostable* instructions [13], it is very unlikely that we have such a problem [11].

At any moment, a probe handler can enable or disable another probe with the *xenprobes_enable()* function. This operation is done by overwriting the probe-point with the breakpoint instruction (in case we want to activate the probe) or the original instruction (in case we want to deactivate the probe).

For both XP and XrP handlers, it is possible to quit the probing loop anytime by calling the framework function *xenprobes_stop()*.

4 Discussion

While the Xenprobes framework is very simple and easy to use, there are some doubts about how much memory is enough for OEA, which place we should put the breakpoints and how to properly access the kernel objects of the probed VM from our handlers. This section is dedicated to discuss these issues.

4.1 OEA Memory Allocation

In our approach, we preallocate an area of memory in the probed VM and use it as OEAs for Xprobes. This can be done thanks to the kernel module *xenprobesU* loaded inside the VM. Regarding the size of this *configurable* area, the developer must anticipate how many probes he wishes to have at the same time. For example, assume that we never use more than *100* probes at once. Each OEA must be able to store one machine instruction and one relative jump instruction, as explained above. In i386, the maximum size of one instruction is *16* bytes, and a relative jump instruction has the fixed size of *5* bytes. Therefore one OEA should have the size of *21* bytes at least. Since we need to have *100* probes, the total size of memory for all the OEAs is *(21 * 100) = 2100* bytes. So in this case, *xenprobesU* preallocates one page of memory, which is *4096* bytes on i386, for OEA at initializing, and that is more than enough for our purpose.

An issue might be raised here: this approach does not allow us extend the memory once we reach the limit of probes. We solve the problem by let Xenprobes notify the probed VM once it sees that the memory is going to run out soon, so *xenprobesU* can allocate a new area of memory for it to use. The notifications regarding new

[13] Instructions such as relative jump, call, software interrupts or that cause hardware side-effects are all *unboostable*.

memory area for more OEAs between Xenprobes and *xenprobesU* are done via XenBus/Xenstore interface.

4.2 Probe Address

Regarding the breakpoints, one of the major concerns is that how can we know exactly where we must put the breakpoints into the probed VM kernel? An intuitive answer for this question is to rely on the kernel source, and and we can decide to put the breakpoints at the addresses corresponding to related lines of source code. Clearly this is a convenient way, because we can inspect the code and see where is the best place to intercept the system flow. So if we know the address in the memory of related lines of code, we can put the breakpoints there. But then, we have another question: how to determine the address of related lines of code?

Fortunately, this problem can be solved quite easily thanks to debugging information coming with kernel binary. In fact, we can exploit a feature made for kernel debugger: If the kernel is compiled with debug option, the kernel binary stores detail information in *DWARF* format about the kernel-types, kernel variables and, most importantly to our purpose, the kernel address of every source code line [6]. As a result, we only need to compile VM's kernel with debug option on, and analyze the kernel binary to get the kernel addresses of the source code lines we want to insert the breakpoints to. Note that this option only generates a big debugged kernel binary file besides the normal kernel binary, and this debugged kernel saves all the information valuable for debugging process. We can still use the normal kernel binary, thus the above requirement does not affect our system at all.

Another choice is to reverse the kernel binary with debugging data, using a tool such as *objdump (1)*. Option *-d* of *objdump* disassembles the machine instructions of the kernel binary, and inform us the virtual address of each instruction, together with the corresponding line of kernel source code. We can investigate the output and easily choose where is the most approriate place to put the probes.

4.3 Accessing VM's Objects

Usually when writing the breakpoint handlers, we want to access to the kernel of the probed VM to inspect its internal status and collect desired information. Here we have a key challenge: how to bridge the semantic gap between the raw memory and kernel objects. To do that, we must be able to have a good knowledge about the OS structure of the VM, so we can have the exact addresses and structures of its kernel objects.

* **Object's address**: In the case of Linux, each global defined object in the kernel is located at a certain memory address, and kept unchanged during its life-time [14]. We can find the address of Linux kernel objects via the kernel symbol file *System.map* coming with the kernel binary.

* **Object structure**: Knowing only the object address is far from enough. For example, in Linux if we want to get the list of kernel modules, first we must retrieve the address of the first kernel module, the global variable *modules*. But then to get the next kernel module pointed by a field named *list.next* in the *module* structure, we must know the relative address of this field in the structure. This job is not trivial, as the *module* structure depends on kernel compiled option, and it might also change between kernel versions [15].

  To extract data about kernel-types, we leverage part of code of LKCD project [15]. LKCD is an open source tool to save and analyze the Linux kernel dump. LKCD can parse the dump thanks to an internal library *libklib*, which extract all the information it needs from the DWARF data in the kernel binary as well as from the kernel symbol file. This library parses the kernel symbols and extracts kernel-types from debugged kernel binary, then caches the data in the memory for its tool named *lcrash* to use. Besides, *libklib* also interprets *lcrash* command, and serves as a disassemble engine for various hardware platforms. Because of these reasons, *libklib* is a very big and complicated code, thus cannot be employed as it is. Another problem is that *libklib* is designed to analyze kernel dump, but not to cope with hostile data. So if somehow the attacker modifies the kernel structure in malicious way, *libklib* might crash.

  In our experiment, we only reused part of *libklib*, in which we only keeps the code that extracts and parses kernel-type information from kernel binary. The library is also hardened to resist potential attacks. Finally, our kernel parse code is around only 14000 lines of C source code, which is about 30% size of the original *libklib*. We plan to include this work into Xenprobes library, so it can be available for all the programs that use our framework.

5 Evaluation

This section presents the performance evaluation of Xenprobes framework comparing with the native speed. The

[14] Note that Linux kernel memory is never swapped out.

[15] Linux kernel never tries to keep compatible between different versions. The Linux kernel developers argue that backward compatibility might block its continuous innovation.

evaluation is done on a para-virtualization Linux VM, the most stable OS platform supported by Xen we have as of this writing. Each benchmark is done in 10 times, and we get the average numbers as the final result.

The configuration of the Xen VMs in the benchmarks are as below:
Dom0: Memory: 384MB RAM, CPU: AthlonXP 2500, IDE HDD: 40GB.
DomU: Memory: 128MB RAM, file-backed swap partition: 512MB, file-backed root partition: 2GB

All the VMs in the tests run Linux Ubuntu distribution (version Breezy Badger).

5.1 Microbenchmark

In the first benchmark, we want to measure how much overhead an XP and XrP can cause to a probed VM. To do that we employ the popular microbenchmark *lmbench* [12]. We inject four probes of XP and XrP into the VM, one at a time, and in the following system-calls: *getppid*, *read*, *write* and *open*. Specifically, we put the handlers at the entry of the *sys_getppid()*, *sys_read()*, *sys_write()*, *sys_open()* functions, respectively. In order to measure the overhead exactly, we use *null* handlers, which are functions doing nothing in the body. We carry out three evaluations: The first, named *Native*, runs the benchmark on the native VM without any probe. The second, named *XP*, registeres the XP with only a null pre-handler. The third evaluation, named *XrP*, registers null entry-handler and return-handler.

Note that since we place the handlers at the entry of these functions, they are always put at *boostable* instruction, so the handling process never suffer a sing-step mode.

The benchmark is done with the commands "*lat_syscall null*", "*lat_syscall read*", "*lat_syscall write*" and "*lat_syscall open*" to measure the overhead on the system-calls *getppid*, *read*, *write* and *open*, respectively. These commands will tell us the latency of these system-calls. Table 1 shows the result of the benchmarks - all the numbers are in microseconds. Next to each number is the overhead compared to the native test, in number of *times*.

	Native	XrP	XP
null	0.2664	107.6731 (404.17)	48.1009 (180.55)
read	0.4732	129.1951 (273.02)	49.6081 (104.83)
write	0.4162	108.8627 (261.56)	49.6027 (119.19)
open	4.0706	117.8936 (28.96)	59.7527 (14.67)

Table 1: Microbenchmark Xenprobes with null handler for XP and XrP.

The benchmarks show us that injecting probes into VM causes quite a big overhead. The *null* benchmark causes the highest penalty for both XrP and XP (*404.17* and *180.55* times, respectively) because *getppid* is a rather simple system-call, thus the main overhead generated is from the switches between the VM, hypervisor and Dom0 when the breakpoint is hit. Meanwhile, the *open* system-call is the most complicated function of all, so the contribute of the penalty by our probes to the overall latency is much more decreased: it is only *28.96* and *14.67* times slower, respectively for XrP and XP.

The notable observation is that in all benchmark, the XrP causes around more than twice overhead compared to the XP. The reason is pretty clear: XrP with two handlers always switches out from its VM twice more than XP with only a pre-handler. In addition, XrP must take time to read and write to the return address of the probed function when the entry function is executed, and when the function returns. These job also causes significant time, as accessing the VM's memory is quite an expensive operation.

5.2 Macrobenchmark

Besides the microbenchmark, we also evaluate Xenprobes in a more reality case with a classical benchmark: decompressing the Linux kernel source. The reason we take this benchmark because Linux kernel contains a lot of data, and the decompress process creates a great number of files and directories. For example unzipping the kernel 2.6.17 generates more than *27000* files and directories, including temporary data. This time we install probes into three system-calls: *mkdir*, *chmod*, *open*. These sytem-calls are triggered when *making directory*, *chmod-ing directories and files*, and *opening files*. These exercises are done quite a lot during unziping kernel: *1201*, *1201* and *19584* times respectively for *mkdir*, *chmod* and *open* [16]. Similarly to the micro benchmark above, this time we also put three XPs (with only null pre-handlers) and three XrPs (with null entry-handlers and return-handlers) , respectively, into the entries of these system-calls.

The benchmark decompresses the Linux kernel 2.6.17 with the command "*time tar xjvf linux-2.6.17.tar.bz2*". Table 2 shows the time to complete the benchmark - all the numbers are in seconds:

	Native	XrP	XP
real	76.781	106.743	81.631
user	44.870	47.360	45.750
sys	5.260	18.380	8.400

Table 2: Macrobenchmark XP and XrP with *mkdir-chmod-open* system-calls and null handlers.

[16] From these numbers, we can safely say that if a new directory is created, it is then immediately *chmod*.

We can see that while the microbenchmark suggests that the probed VM causes very high overhead, in reality the impact is not that much: XP evaluation causes only around *6.31%* penalty, and XrP evaluation causes around *39.02%* overhead. Again, this benchmark shows that probing with XP is significantly faster than XrP. Another observation is that the system mainly suffers in kernel execution, but not in user-space.

In another attempt to measure the impact when more probes are used and put at more performance critical places, we run another test. This time, along with three probes in the above benchmark, we put two more probes (XPs and XrPs, respectively) into the *read* and *write* system-calls. These system-calls are executed in a great number of times when the kernel is unzipped: *78011* times and *139981* times for *read* and *write*, respectively.

The kernel unzip benchmark gives us the result below, in Table 3. All the numbers are in seconds.

	Native	**XrP**	**XP**
real	76.781	165.187	94.572
user	44.870	45.050	44.930
sys	5.260	28.800	16.000

Table 3: Macrobenchmark XP and XrP with *read-write-mkdir-chmod-open* system-calls and null handlers.

Again, we can confirm that even with probes placed at critical execution path in kernel, the performance penalty is not too high. Especially, this benchmark shows the major improvements of XP against XrP: *23.17%* overhead compares with *115.14%* overhead.

Our conclusion is that Xenprobes can be employed to inspect VM's status at run-time without causing too much overhead.

6 Related Works

Our work is strongly inspired by Kprobes, a probing framework available in Linux kernel. Kprobes is widely used for kernel tracing [13] and performance evaluation.

When using Xenprobes for the same job on Xen VMs, our framework has some advantages over Kprobes as mentioned in section 1. Besides, Xenprobes brings several benefits as followings.

- When programming Xenprobes handler, we do not need to worry about page-fault problem, as we work in user-space of Dom0. Kprobes handlers must not cause other exceptions such as page-fault. Though Kprobes allows to specify an user-define page-fault handler to handle the issue, the specified page-fault handler cannot always solve the problem. The Kprobes developers are still working to make the Kprobes fault handling more robust.
- Xenprobes has no problem of reentry as with Kprobes [8].

However, Xenprobes suffers a drawback comparing with Kprobes: Kprobes works in Linux and can interact with the real hardware, thus can be used to monitor and debug all kind of hardware devices. Because Xen VMs cannot work with real hardware, but use the virtual hardware provided by hypervisor, Xenprobes cannot be used to debug arbitrary device drivers. Actually this is a fundamental issue of Xen, rather than Xenprobes.

Another notable difference with Kprobes is that besides "normal" probe, Kprobes supports two other types: Jprobes and Function-return probes [11]. Jprobes is mainly used to gather the probed function parameters, and Function-return probes is used to get the returned value of probed function. Actually from our observation, it is quite common for developers to employ both of these two type of probes at the same time. Therefore, we strongly believe that it is better to combine them, and Xenprobes framework realizes our idea: it is possible to collect both function parameters and returned value with only one XrP probe.

Technically, XrP is inspired by an old version of function-return probes of Kprobes: Until Linux kernel version 2.6.16, Kprobes also employs the "trampoline" technique with two Kprobes to handle the function at return time. But from version 2.6.17, Kprobes uses a new technique, in which the probe for the trampoline is eliminated by some assembly code that saves and recovers the return address. The reason we adopt the old technique of Kprobes for XrP is that because we need two switches to Dom0: first is to execute the entry-handler, and second is to execute the return-handler.

Xenprobes exploits the debugging architecture introduced by Xen. Xen also has built-in support for debugging VM kernel at run-time with gdb [1]. However, as we discussed in the first section, debugging tool does not allow automatic monitoring and probing VM. That is the gap our framework tries to fill in.

Our work shares some ideas with the work of K.Arigos et.al in [3]: their paper also proposes to put breakpoints into Xen VMs to get notified when interested events occur. However, the way we handle debugging events is quite different from [3]: their proposal pushes the breakpoint handlers into the hypervisor layer, and loads the handling policy from Dom0 to hypervisor via an add-in hypercall. Their idea is to let the hypervisor capture the breakpoint events and analyze them there. In order to do that, the authors made quite a big modification to the hypervisor layer (around 2700 lines of code), which they also mentioned as belong to the *Trusted Computing Base* (TCB), the critical and core component required to enforce the system security. We would argue that it is not

desired to make such a major change to such an important component, because it makes the whole system less stable, less secure as well as increase the maintenance cost [17]. In our solution, Xenprobes takes the advantage of the Xen debugging technique, thus makes absolutely no modification to the hypervisor.

Another disadvantage of putting all breakpoint handlers inside the hypervisor is that every time we wish to modify the probe handlers, we have to to alter then recompile the hypervisor, and the system must be rebooted for the change to take effect. Meanwhile, Xenprobes puts all the handlers in user-space, which makes them very easy and convenient to work with. All the development are done in Dom0, and requires no recompilation or reboot whatsoever to the hypervisor.

Last but not least, we go further in providing a framework for injecting breakpoints into the VM, so it can be employed by other projects, not only for security purpose. We are going to publish the code of Xenprobes under open source license (GPL) for everybody to use.

7 Conclusions

This paper describes Xenprobes, a novel framework that allows developers to probe Xen VM's kernel in a more convenient way. In designing Xenprobes, we exploit the infrastructure available for Xen debugging architecture, so the framework offers several interesting benefits in a very simple and easy-to-use interface. Most importantly, Xenprobes allows developers to program their probe handlers in user-space, and it is possible to probe multiple VM at the same time. Because our approach is independent of OS, all the OS-es are supported, even closed source ones such as Microsoft Windows.

Regarding the performance penalty caused by Xenprobes, we believe that the impact is acceptable and can be used in production systems.

Xenprobes internal has quite many things in common with Kprobes, so we believe that it can be easily adopted by the developers who are currently familiar with Kprobes.

We are going to release Xenprobes under the open source GPL license, with the hope that it can attract more interests and become useful for many people.

While the technique to probe guest machine in this paper is specifically described on Xen environment, there is no reason why it does not work on other kind of virtual machine. We are working to have a similar framework on *KVM* [9], a virtual machine technology based on virtualization-enable processor [18].

[17] Xen layer can change anytime, and actually always under active development as of this writing.

[18] KVM has been merged into Linux kernel since version 2.6.20.

References

[1] A.KAMBLE, N., NAKAJIMA, J., AND K.MALLICK, A. Evolution in kernel debugging using hardware virtualization with Xen. In *Proceedings of the 2006 Ottawa Linux Symposium* (Ottawa, Canada, July 2006).

[2] AMD CORP. AMD64 Architecture Programmer's Manual Volume 2: System Programming. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf`, 2005.

[3] ASRIGO, K., LITTY, L., , AND LIE, D. Virtual machine-based honeypot monitoring. In *Proceedings of the 2nd international conference on Virtual Execution Environments* (New York, NY, USA, June 2006), ACM Press.

[4] CLARK, B., DESHANE, T., DOW, E., EVANCHIK, S., FINLAYSON, M., HERNE, J., AND MATTHEWS, J. N. Xen and the art of repeated research. In *Proceedings of the Usenix annual technical conference, Freenix track.* (July 2004), pp. 135–144.

[5] DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., PRATT, I., WARFIELD, A., BARHAM, P., AND NEUGEBAUER, R. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles* (October 2003).

[6] DWARF WORKGROUP. DWARF Debugging Format Standard. `http://dwarf.freestandards.org/Home.php`, January 2006.

[7] INTEL CORP. Intel Virtualization Technology. `http://www.intel.com/technology/virtualization/index.htm`, 2006.

[8] JIM KENISTON AND PRASANNA PANCHAMUKHI. Kprobes Documentation. `linux-kernel/Documentation/kprobes.txt`, October 2006.

[9] KVM PROJECT. KVM: Kernel based Virtual Machine. `http://kvm.qumranet.com`, 2006.

[10] LINSYSSOFT TECHNOLOGIES LTD. KGDB: Linux kernel source level debugger. `http://kgdb.linsyssoft.com/`, 2006.

[11] MAVINAKAYANAHALLI, A., PANCHAMUKHI, P., AND KENISTON, J. Probing the Guts of Kprobes. In *Proceedings of The Linux Symposium 2006* (July 2006).

[12] MCVOY, L., AND STAELIN, C. LMbench - Tools for Performance Analysis. `http://lmbench.sf.net`, August 2004.

[13] PANCHAMUKHI, P. Kernel debugging with Kprobes. `http://www-128.ibm.com/developerworks/library/l-kprobes.html`, 2004.

[14] PRATT, I., FRASER, K., HAND, S., LIMPACH, C., WARFIELD, A., MAGENHEIMER, D., NAKAJIMA, J., AND MALLICK, A. Xen 3.0 and the art of virtualization. In *Proceedings of the 2005 Ottawa Linux Symposium* (Ottawa, Canada, July 2005).

[15] SGI INC. LKCD - Linux Kernel Crash Dump. `http://lkcd.sf.net`, April 2006.

[16] SILICON GRAPHIC INC. KDB: Built-in kernel debugger. `http://oss.sgi.com/projects/kdb/`, 2006.

[17] XEN PROJECT. Xen interface manual. `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/interface/interface.html`, August 2006.

Virtual Machine Memory Access Tracing With Hypervisor Exclusive Cache*

Pin Lu and Kai Shen
Department of Computer Science, University of Rochester
{pinlu, kshen}@cs.rochester.edu

Abstract

Virtual machine (VM) memory allocation and VM consolidation can benefit from the prediction of VM page miss rate at each candidate memory size. Such prediction is challenging for the hypervisor (or VM monitor) due to a lack of knowledge on VM memory access pattern. This paper explores the approach that the hypervisor takes over the management for part of the VM memory and thus all accesses that miss the remaining VM memory can be transparently traced by the hypervisor.

For online memory access tracing, its overhead should be small compared to the case that all allocated memory is directly managed by the VM. To save memory space, the hypervisor manages its memory portion as an exclusive cache (*i.e.*, containing only data that is not in the remaining VM memory). To minimize I/O overhead, evicted data from a VM enters its cache directly from VM memory (as opposed to entering from the secondary storage). We guarantee the cache correctness by only caching memory pages whose current contents provably match those of corresponding storage locations. Based on our design, we show that when the VM evicts pages in the LRU order, the employment of the hypervisor cache does not introduce any additional I/O overhead in the system.

We implemented the proposed scheme on the Xen para-virtualization platform. Our experiments with microbenchmarks and four real data-intensive services (SPECweb99, index searching, TPC-C, and TPC-H) illustrate the overhead of our hypervisor cache and the accuracy of cache-driven VM page miss rate prediction. We also present the results on adaptive VM memory allocation with performance assurance.

1 Introduction

Virtual machine (VM) [2,8,22] is an increasingly popular service hosting platform due to its support for fault containment, performance isolation, ease of transparent system management [4] and migration [7]. For data-intensive services, the problem of VM memory allocation arises in the context of multi-VM memory sharing and service consolidation. To achieve performance isolation with quality-of-service (QoS) constraints, it is desirable to predict the VM performance (or page miss rate) at each candidate memory size. This information is also called page miss ratio curve [27].

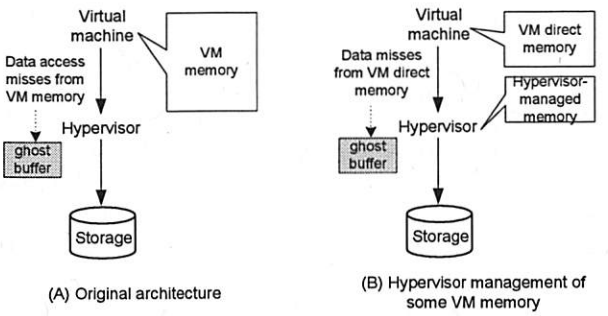


Figure 1: Virtual machine architecture on which the hypervisor manages part of the VM memory. Less VM direct memory results in more hypervisor-traceable data accesses.

Typically, the hypervisor (or the VM monitor) sees all VM data accesses that miss the VM memory in the form of I/O requests. Using the ghost buffer [12, 17] technique, it can predict the VM page miss rate for memory sizes beyond its current allocation. However, since data accesses that hit the VM memory are not visible to the hypervisor, it is challenging to estimate VM page miss rate for memory sizes smaller than its current allocation. An intuitive idea to predict more complete VM page miss rate information is the following (illustrated in Figure 1). The hypervisor takes over the management for part of the VM memory and thus all accesses that miss the remaining VM directly-managed memory (or *VM direct memory*) can be transparently traced by the hypervisor. By applying the same ghost buffer technique, the hypervisor can now predict VM performance for memory sizes beyond the VM direct memory size.

To be able to apply online, our VM memory access tracing technique must be efficient. More specifically, the hypervisor memory management should deliver competitive performance compared to the original case that all allocated VM memory is directly managed by the VM OS. In order to avoid double caching, we keep the hypervisor memory as an exclusive cache to the VM direct

*This work was supported in part by the National Science Foundation (NSF) grants CCR-0306473, ITR/IIS-0312925, CNS-0615045, CCF-0621472, NSF CAREER Award CCF-0448413, and an IBM Faculty Award.

memory. Exclusive cache [5,12,25] typically admits data that is just evicted from its upper-level cache in the storage hierarchy (VM direct memory in our case). It is efficient for evicted VM data to enter directly into the hypervisor cache (as opposed to loading from secondary storage). However, this may introduce caching errors when the VM memory content does not match that of corresponding storage location. We ensure the correctness of our cache by only admitting data that has provably the same content as in the corresponding storage location. We achieve this by only accepting evicted pages before reuse to avoid data corruptions, and by maintaining two-way mappings between VM memory pages and storage locations to detect mapping changes in either direction.

Based on our design, our hypervisor exclusive cache is able to manage large chunk of a VM's memory without increasing the overall system page faults (or I/O overhead). However, the cache management and minor page faults (*i.e.*, data access misses at the VM direct memory that subsequently hit the hypervisor cache) incur some CPU overhead. We believe that the benefit of predicting accurate VM page miss ratio curve outweighs such overhead in many situations, particularly for data-intensive services where I/O is a more critical resource than CPU. Additionally, when the hypervisor cache is employed for acquiring VM memory access pattern and guiding VM memory allocation, it only needs to be enabled intermittently (when a new memory allocation is desired).

2 Related Work

VM memory allocation Virtual machine (VM) technologies like Disco [3, 9], VMware [8, 22, 23], and Xen [2] support fault containment and performance isolation by partitioning physical memory among multiple VMs. It is inherently challenging to derive good memory allocation policy at the hypervisor due to the lack of knowledge on VM data access pattern that is typically available to the OS. Cellular Disco [9] supports memory borrowing from cells rich of free memory to memory-constrained cells. However, cells in their context are VM containers and they are more akin to physical machines in a cluster. Their work does not address policy issues for memory allocation among multiple VMs within a cell.

In VMware ESX server, Waldspurger proposed a sampling scheme to transparently learn the proportion of VM memory pages that are accessed within a time period [23]. This result can be used to derive a working set estimation and subsequently to guide VM memory allocation. This sampling approach requires very little overhead but it is less powerful than VM memory allocation based on accurate VM page miss ratio curve.

1. The sampling approach may not be able to support memory allocation with flexible QoS constraint. One possible allocation objective is to minimize a system-wide page miss rate metric with the constraint that no VM may have more than $\delta\%$ increase in page misses compared to its baseline allocation.
2. Although the sampling approach can estimate the amount of memory accessed within a period of time (*i.e.*, the working set), the working set may not always directly relate to the VM's performance behavior. For example, it is known that the working set model may over-estimate the memory need of a program with long sequential scans [6, 18].

We will elaborate on these issues in Section 4.2 and experimentally demonstrate them in Section 6.5.

Exclusive cache management Wong and Wilkes [25] argued that exclusive lower-level caches are more effective than inclusive ones (by avoiding double caching). This is particularly the case when lower-level caches are not much larger than upper-level ones in the storage hierarchy. To implement an exclusive cache, they introduced a DEMOTE operation to notify lower-level caches about data evictions from upper-level caches. To achieve cache correctness, they assume the evicted data contains exactly the same content as in the corresponding storage location. They do not address how this is achieved in practice.

Chen *et al.* [5] followed up Wong and Wilkes's work by proposing a transparent way to infer upper-level cache (memory page) evictions. By intercepting all I/O reads/writes, they maintain the mapping from memory pages to storage blocks. A mapping change would indicate a page reuse which infers an eviction has occurred earlier. Jones *et al.* [12] further strengthened the transparent inference of page reuses by considering additional issues such as storage block liveness, file system journaling, and unified caches (virtual memory cache and file system buffer cache). In these designs, the exclusive cache is assumed to be architecturally closer to the lower-level storage devices and data always enters the cache from the storage devices. In our context, however, it is much more efficient for evicted data to enter the hypervisor cache directly from VM memory. This introduces potential correctness problems when the entering VM memory content does not match the storage content. In particular, both Chen *et al.* [5] and Jones *et al.* [12] detect page reuses and then infer earlier evictions. At page reuse time, the correct content for the previous use may have already been zero-cleaned or over-written — too late for loading into the hypervisor cache.

Hypervisor-level cache As far as we know, existing hypervisor-level buffer cache (*e.g*, Copy-On-Write disks in Disco [3] and XenFS [24]) is used primarily for the purpose of keeping single copy of data shared across

multiple VMs. At the absence of such sharing, one common belief is that buffer cache management is better left to the VM OS since it knows more about the VM itself. In this paper, we show that the hypervisor has sufficient information to manage the cache efficiently since all accesses to the cache are trapped in software. More importantly, the cache provides a transparent means to learn the VM data access pattern which in turn guides performance-assured memory allocation.

OS-level program memory need estimation Zhou *et al.* [27] and Yang *et al.* [26] presented operating system (OS) techniques to estimate program memory need for achieving certain desired performance. The main idea of their techniques is to revoke access privilege on (infrequently accessed) partial program memory and trap all accesses on this partial memory. Trapped data accesses are then used to estimate program page miss ratio or other performance metric at different memory sizes. While directly applying these OS-level technique within each VM can estimate VM memory need, our hypervisor-level approach attains certain advantages while it also presents unique challenges.

- *Advantages*: In a VM platform, due to potential lack of trust between the hypervisor and VMs, it is more appropriate for the hypervisor to collect VM memory requirement information rather than let the VM directly report such information. Further, given the complexity of OS memory management, separating the memory need estimation from the OS improves the whole system modularity.
- *Challenges*: Correctly maintaining the hypervisor exclusive cache is challenging due to the lack of inside-VM information (*e.g.*, the mapping information between memory pages and corresponding storage locations). Such information is readily available for an OS-level technique within each VM. Further, the employment of a hypervisor buffer cache potentially incurs more management overhead. More careful design and implementation are needed to keep such overhead small.

3 Hypervisor-level Exclusive Cache

Our hypervisor-level cache has several properties that are uncommon to general buffer caches in storage hierarchies. First, its content is exclusive to its immediate upper-level cache in the storage hierarchy (VM direct memory). Second, this cache competes for the same physical space with its immediate upper-level cache. Third, data enters this cache directly from its immediate upper-level cache (as opposed to entering from its immediate lower-level cache in conventional storage systems). These properties combined together present unique challenges for our cache design. In this section, we present the basic design of our hypervisor-level exclusive cache, propose additional support to ensure cache correctness, discuss the transparency of our approach to the VM OS, and analyze its performance (cache hit rate) and management overhead.

3.1 Basic Design

The primary design goals of our hypervisor cache are that: 1) it should try not to contain any data that is already in VM memory (or to be exclusive); and 2) it should try to cache data that is most likely accessed in the near future. To infer the access likelihood of a page, we can use the page eviction order from the VM as a hint. This is because the VM OS would only evict a page when the page is believed to be least useful in the near future.

For write accesses, we can either support delayed writes (*i.e.*, writes are buffered until the buffered copies have to be evicted) or write-through in our cache management. Delayed writes reduce the I/O traffic, however delayed writes in the hypervisor cache are not persistent over system crashes. They may introduce errors over system crashes when the VM OS counts the write completion as a guarantee of data persistence (*e.g.*, in file system `fsync()`). Similar problems were discussed for delayed writes in disk controller cache [16]. On the other hand, the SCSI interface allows the OS to individually specify I/O requests with persistence requirement (through the force-unit-access or FUA bit). In general, we believe delayed writes should be employed whenever possible to improve performance. However, write-through might have to be used if persistence-related errors are not tolerable and we cannot distinguish those writes with persistence requirement and those without.

In our cache management, all data units in the hypervisor cache (typically memory pages) are organized into a queue. Below we describe our management policy, which defines actions when a read I/O request, a VM data eviction, or a write I/O request (under write-through or delayed write) reaches the hypervisor. A simplified illustration is provided in Figure 2.

- Read I/O request $\Longrightarrow$ If it hits the hypervisor cache, then we bring the data from the hypervisor cache to VM memory and return the I/O request. To avoid double caching at both levels, we move it to the queue head (closest to being discarded) in the hypervisor cache or explicitly discard it. If the request misses at the hypervisor cache, then we bring the data from external storage to the VM memory as usual. We do *not* keep a copy in the hypervisor cache.
- VM data eviction $\Longrightarrow$ We cache the evicted data at the queue tail (furthest away from being discarded)

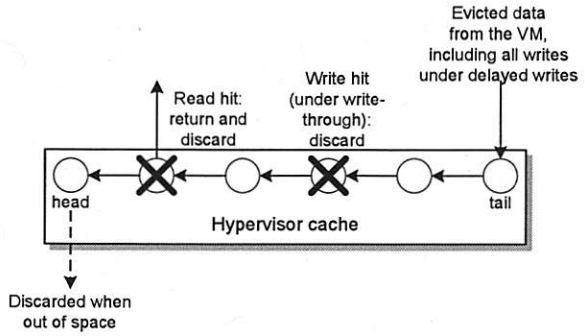


Figure 2: A simplified illustration of our hypervisor cache management.

of the hypervisor cache. In this way, the hypervisor cache discards data in the same order that the VM OS evicts them. If the hypervisor cache and the VM memory are not strictly exclusive, it is also possible for the evicted data to hit a cached copy. In this case, we simply move the cached copy to the queue tail.

- Write I/O request (write-through) $\Longrightarrow$ We write the data to the external storage as usual. If the request hits the hypervisor cache, then we also discard the hypervisor cache copy. We do not need to keep an updated copy since the VM memory should already contain it.
- Write I/O request (delayed write) $\Longrightarrow$ Each write I/O request is buffered at the hypervisor cache (marked as dirty) and then the request returns. The data is added at the hypervisor cache queue tail. If the request hits an earlier cached unit on the same storage location, we discard that unit. Although the write caching creates temporary double buffering in VM memory and hypervisor cache, this double buffering is of very short duration if the VM OS also employs delayed writes (in this case a write is typically soon followed by an eviction). Dirty cached data will eventually be written to the storage when they reach the queue head to be discarded.

To support lookup, cached entries in the hypervisor cache are indexed according to their mapped storage locations. Therefore we need to know the mapped storage location for each piece of evicted data that enters the cache. Such mapping can be constructed at the hypervisor by monitoring I/O requests between VM memory pages and storage locations [5]. Specifically, an I/O (read or write) between page p and storage location s establishes a mapping between them (called *P2S* mapping). A new mapping for a page replaces its old mapping. Additionally, we delete a mapping when we are aware that it becomes stale (*e.g.*, when a page is evicted or released). We distinguish page release from page eviction in that page eviction occurs when the VM runs out of memory space while page release occurs when the VM OS

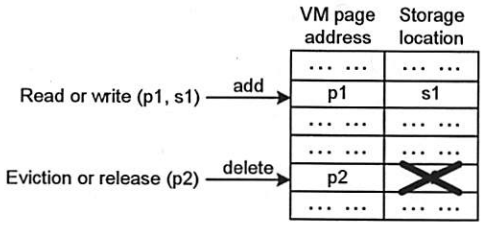


Figure 3: An illustration of the VM page to storage location mapping table (or P2S table in short).

feels it is not useful. For example, a page in file system buffer cache is released when its mapped storage location becomes invalid (*e.g.*, as a result of file truncation). Although both page eviction and release should delete the page to storage location mapping, only evicted data should enter the hypervisor cache. Figure 3 provides an illustration of the VM page to storage location mapping table.

3.2 Cache Correctness

Since the hypervisor cache directly supplies data to a read request that hits the cache, the data must be exactly the same as in the corresponding storage location to guarantee correctness. To better illustrate our problem, below we describe two realistic error cases that we experienced:

Missed eviction/release: Page p is mapped to storage location s and the hypervisor is aware of this mapping. Later, p is reused for some other purpose but the hypervisor fails to detect the page eviction or release. Also assume this reuse slips through the detection of available reuse detection techniques (*e.g.*, Geiger [12]). When p is evicted again and the hypervisor captures the eviction this time, we would incorrectly admit the data into the cache with mapped storage location s. Later read of s will hit the cache and return erroneous data.

Stale page mapping: The VM OS may sometimes keep a page whose mapping to its previously mapped storage location is invalid. For instance, we observe that in the Linux 2.6 ext3 file system, when a meta-data block is recycled (due to file deletion for example), its memory cached page would remain (though inaccessible from the system). Since it is inaccessible (as if it is a leaked memory), its consistency does not need to be maintained. At its eviction time, its content may be inconsistent with the storage location that it was previously mapped to. Now if we cache them, we may introduce incorrect data into the hypervisor cache. The difficulty here is that without internal OS knowledge, it is hard to tell whether an evicted page contains a stale page mapping or not.

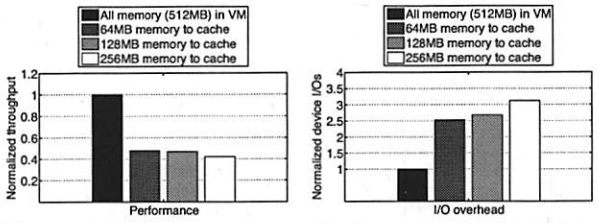


Figure 4: Service throughput reduction and real device I/O increase of the hypervisor cache management when new cached data is always loaded from the storage. Results are for a trace-driven index searching workload (described in Section 6.1). Due to such large cost, our cache management allows evicted VM data to enter the cache directly from VM memory.

We identify two sufficient conditions that would meet our cache correctness goal:

Condition 1 (admission): Each time we cache a page, we guarantee it contains the same content as in the corresponding storage location.

Condition 2 (invalidation): The hypervisor captures all I/O writes that may change storage content. If we have a cached copy for the storage location being written to, we remove it from the hypervisor cache.

The invalidation condition is easy to support since the hypervisor typically captures all I/O requests from VMs to storage devices. The challenge lies in the support for the admission condition due to the limited amount of information available at the hypervisor. Note that the admission condition is trivially satisfied if we always load new data from the storage [5,12]. However, this is inappropriate in our case due to its large cost. To illustrate the cost when data enters the cache from the storage, Figure 4 shows the substantial performance reduction and I/O increase for a trace-driven index searching workload.

Our correctness guarantee is based on a set of assumptions about the system architecture and the VM OS. First, we assume that the hypervisor can capture all read/write I/O operations to the secondary storage. Second, we assume the hypervisor can capture every page eviction and release (before reuse) in the VM. Third, we assume no page is evicted from the VM while it is dirty (*i.e.*, it has been changed since last time it was written to the storage). While the first and the third assumptions are generally true without changing the VM OS, the second assumption needs more discussion and we will provide that later in Section 3.3.

In addition to these assumptions, we introduce a reverse mapping table that records the mapping from storage locations to VM pages (or S2P table). Like the P2S mapping table described in Section 3.1, a mapping is established in the S2P table each time a read/write I/O request is observed at the hypervisor. A new mapping for a storage location replaces its old mapping. Each time a page p is evicted from the VM memory, we check the two mapping tables. Let p be currently mapped to storage location s in the P2S table and s be mapped to page p' in the S2P table. We admit the evicted page into the hypervisor cache only if $p = p'$. This ensures that p is the last page that has performed I/O operation on storage location s.

Correctness proof: We prove that our approach can ensure the admission condition for correctness. Consider each page p that we admit into the hypervisor cache with a mapped storage location s. Since we check the two mapping tables before admitting it, the most recent I/O (read or write) that concerns p must be on s and the reverse is also true. This means that the most recent I/O operation about p and the most recent I/O operation about s must be the same one. At the completion of that operation (no matter whether it is a read or write), p and s should contain the same content. Below we show that neither the storage content nor the page content has changed since then. The storage content has not changed since it has not established mapping with any other page (otherwise the S2P table would have shown it is mapped to that page). The page content has not changed because it has not been reused and it is not dirty. It is not reused since otherwise we should have seen its eviction or release before reuse and its mapping in the P2S table would have been deleted. Note our assumption that we can capture every page eviction and release in the VM. ■

3.3 Virtual Machine Transparency

It is desirable for the hypervisor cache to be implemented with little or no change to the VM OS. Most of our design assumptions are readily satisfied by existing OSes without change. The only non-transparent aspect of our design is that the hypervisor must capture every page eviction and release (before reuse) in the VM. A possible change to the VM OS is to make an explicit trap to the hypervisor at each such occasion. The only information that the trap needs to provide is the address of the page to be evicted or released.

The suggested change to the VM OS is semantically simple and it should be fairly easy to make for existing OSes. Additionally, the eviction or release notification should not introduce additional fault propagation vulnerability across VM boundaries. This is because the only way this operation can affect other VMs' correctness is when multiple VMs are allowed to access the same storage locations. In this case a VM can always explicitly write invalid content into these shared locations. In summary, our suggested change to the VM OS fits well into a para-virtualization platform such as Xen [2].

We also provide some discussions on the difficulty of implementing a hypervisor-level exclusive cache in a

fully transparent way. Earlier transparent techniques [5, 12] can detect the eviction of a page after its reuse. However, reuse time detection is too late for loading evicted data directly from VM memory to the hypervisor cache. At reuse time, the original page content may have already be changed and some OSes would have zeroed the page before its reuse. Further, it is not clear any available transparent technique can capture every page reuse without fail (no false negative).

3.4 Performance and Management Overhead

The primary goal of the hypervisor cache is to allow transparent data access tracing. Yet, since it competes for the same physical space with the VM direct memory, its employment in an online system should not result in significant VM performance loss. This section analyzes the cache hit rate and management overhead of our hypervisor cache scheme compared to the original case in which all memory is directly managed by the VM.

Caching performance We compare the overall system cache hit rate of two schemes: the first contains a VM memory of X pages with an associated hypervisor cache of Y pages (called *Hcache* scheme); the other has a VM memory of $X+Y$ pages with no hypervisor cache (called *VMonly* scheme). Here we consider an access to be a hit as long as it does not result in any real device I/O. We use an ideal model in which the VM OS employs perfect LRU cache replacement policy. Under this model, we show that Hcache and VMonly schemes achieve the same cache hit rate on any given data access workload. Our result applies to both read and write accesses if we employ delayed writes at the hypervisor cache. Otherwise (if we employ write-through) the result only applies to reads.

Consider a virtual LRU stack [15] that orders all pages according to their access recency — a page is in the k-th location from the top of the stack if it is the k-th most recently accessed page. At each step of data access, the VM memory under the VMonly scheme contains the top $X + Y$ pages in the virtual LRU stack. For the Hcache scheme, the top X pages in the stack are in the VM memory while the next Y pages should be in the hypervisor cache. This is because our hypervisor cache is exclusive to the VM memory and it contains the most recently evicted pages from the VM memory (according to the cache management described in Section 3.1). So the aggregate in-memory content is the same for the two schemes at each step of data access (as shown in Figure 5). Therefore VMonly and Hcache should have the identical data access hit/miss pattern for any given workload and consequently they should achieve the same cache hit rate.

The above derivation assumes that the hypervisor cache is strictly exclusive to the VM memory and all evicted pages enter the hypervisor cache. These assumptions may not be true in practice for the following reasons. First, there is a lag time between a page being added to the hypervisor and it is being reused in the VM. The page is doubly cached during this period. Second, we may prevent some evicted pages from entering the hypervisor cache due to correctness concern. However, these exceptions are rare in practice so that they do not visibly affect the cache hit rate (as demonstrated by our experimental results in Section 6.3).

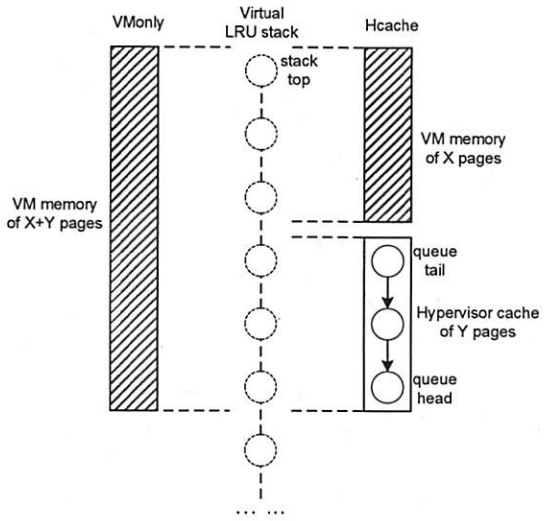


Figure 5: In-memory content for VMonly and Hcache when the VM OS employs perfect LRU replacement.

Management overhead The employment of the hypervisor cache introduces additional management (CPU) overhead, including the mapping table lookup and simple queue management in the hypervisor cache. Additionally, the handling of page eviction and minor page fault (*i.e.*, data access misses at the VM memory that subsequently hit the hypervisor cache) requires data transfer between the VM memory and the hypervisor cache. Page copying can contribute a significant amount of overhead. Remapping of pages between the VM and the hypervisor cache may achieve the goal of data transfer with much less overhead. Note that for minor page faults, there may also be additional CPU overhead within the VM in terms of page fault handling and I/O processing (since this access may simply be a hit in VM memory if all memory is allocated to the VM).

For the employment of the hypervisor cache, the benefit of transparent data access is attained at the additional CPU cost. More specifically, when we move away Y pages from the VM memory to the hypervisor cache, we can transparently monitor data accesses on these pages while at the same time we may incur overhead of cache management and minor page faults on them.

Note that the monitoring of these Y pages provides more useful information than monitoring Y randomly chosen pages [23]. This is because the pages in the hypervisor cache are those that the VM OS would evict first when its memory allocation is reduced. Access statistics on these pages provide accurate information on additional page misses when some memory is actually taken away.

4 Virtual Machine Memory Allocation

With the hypervisor management for part of the VM memory, we discuss our ability to predict more complete VM page miss ratio curve and consequently to guide VM memory allocation. We then provide an example of complete-system workflow for our guided VM memory allocation. We also discuss a potential vulnerability of our adaptive memory allocation to VM manipulation.

4.1 VM Miss Ratio Curve Prediction

To best partition the limited memory for virtual machines (VMs) on a host or to facilitate VM consolidation over a cluster of hosts, it is desirable to know each VM's performance or page miss rate at each candidate allocation size (called miss ratio curve [27]). Jones *et al.* [12] showed that the miss ratio curve can be determined for memory sizes larger than the current memory allocation when all I/O operations and data evictions of the VM are traced or inferred. Specifically, the hypervisor maintains a ghost buffer (a simulated buffer with index data structure but no actual page content) [17]. Ghost buffer entries are maintained in the LRU order and hypothetical hit counts on each entry are tracked. Such hit statistics can then be used to estimate the VM page hit rate when the memory size increases (assuming the VM employs LRU page replacement order). To reduce ghost buffer statistics collection overhead, hit counts are typically maintained on segments of ghost buffer pages (*e.g.*, 4 MB) rather than on individual pages.

Our hypervisor cache-based scheme serves as an important complement to the above VM miss ratio prediction. With the hypervisor management for part of the VM memory, we can transparently trace all VM data accesses that miss the remaining VM direct memory. This allows the hypervisor to apply the ghost buffer technique to predict VM page miss rate at all memory sizes beyond the VM direct memory size (which is smaller than the currently allocated total VM memory size). In particular, this approach can predict the amount of performance loss when some memory is taken away from the VM.

4.2 Memory Allocation Policies

With known miss ratio curve for each VM at each candidate memory allocation size, we can guide multi-VM memory allocation with flexible QoS constraint and strong performance assurance. Let each VM on the host start with a baseline memory allocation, the general goal is to adjust VM memory allocation so that the overall system-wide overall page misses is reduced while certain performance isolation is maintained for each VM. Within such a context, we describe two specific allocation policies. The purpose of the first policy is to illustrate our scheme's ability in supporting flexible QoS constraint (that a sampling-based approach is not capable of). The second policy is a direct emulation of a specific sampling-based approach (employed in the VMware ESX server [23]) with an enhancement.

Isolated sharing We dynamically adjust memory allocation to the VMs with the following two objectives:

- *Profitable sharing*: Memory is divided among multiple VMs to achieve low system-wide overall page misses. In this example, we define the system-wide page miss metric as the geometric mean of each VM's miss ratio (its number of page misses under the new memory allocation divided by that under its baseline allocation). Our choice of this metric is not necessary. We should also be able to support other system-wide performance metrics as long as they can be calculated from the predicted VM miss ratio curves.
- *Isolation constraint*: If a VM's memory allocation is less than its baseline allocation, it should have a bounded performance loss (*e.g.*, no more than $\delta\%$ in additional page misses) compared to its performance under the baseline allocation.

We describe our realization of this allocation policy. One simple method is to exhaustively check all candidate allocation strategies for estimated system-wide performance metric and individual VM isolation constraint compliance. The computation overhead for such a method is typically not large for three or fewer VMs on a host. With three VMs sharing a fixed total memory size, there are two degrees of freedom in per-VM memory allocation. Assuming 100 different candidate memory sizes for each VM, around 10,000 different whole-system allocation strategies need to be checked.

The search space may become too large for exhaustive checking when there are four or more VMs on the host. In such cases, we can employ a simple greedy algorithm. At each step we try to move a unit of memory (*e.g.*, 4 MB) from the VM with the least marginal performance loss to the VM with the largest marginal performance gain if the adjustment is considered profitable (it reduces the estimated system-wide page miss metric). The VM that loses memory must still satisfy the isolation constraint at its new allocation. The algorithm stops

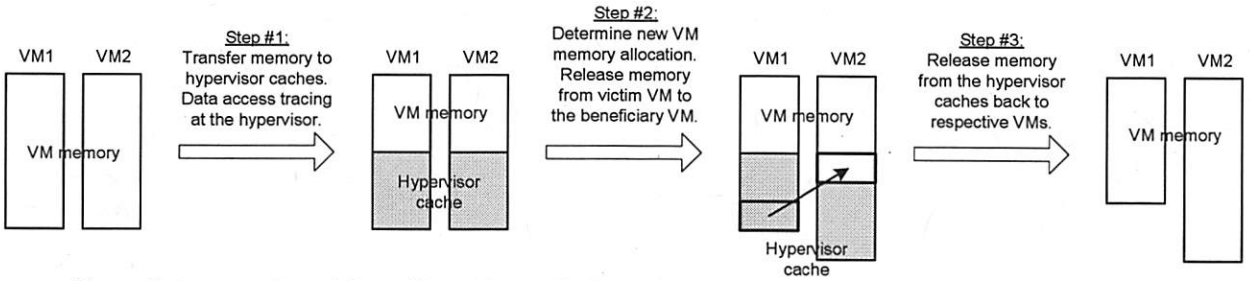


Figure 6: An example workflow of hypervisor cache-based data access tracing and multi-VM memory allocation.

when all profitable adjustments would violate the isolation constraint. Other low-overhead optimization algorithms such as simulated annealing [13] may also be applied in this case. Further exploration falls beyond the scope of this paper.

Note that in all the above algorithms, we do not physically move memory between VMs when evaluating different candidate allocation strategies. For each allocation strategy, the performance (page miss rate) of each VM can be easily estimated by checking the predicted VM page miss ratio curve.

VMware ESX server emulation The VMware ESX server employs a sampling scheme to estimate the amount of memory accessed within a period of time (*i.e.*, the working set). Based on the working set estimation, a VM whose working set is smaller than its baseline memory allocation is chosen as a *victim* — having some of its unneeded memory taken away (according to an idle memory tax parameter). Those VMs who can benefit from more memory then divide the surplus victim memory (according to certain per-VM share).

We can easily emulate the above memory allocation policy. Specifically, our per-VM page miss ratio curve prediction allows us to identify victim VMs as those whose performance does not benefit from more memory and does not degrade if a certain amount of memory is taken away. Since the VM page miss ratio curve conveys more information than the working set size alone does, our scheme can identify more victim VMs appropriately. Consider a VM that accesses a large amount of data over time but rarely reuses any of them. It would exhibit a large working set but different memory sizes would not significantly affect its page fault rate. Such problems with the working set model were well documented in earlier studies [6, 18].

4.3 An Example of Complete-System Workflow

We describe an example of complete-system workflow. We enable the hypervisor cache when a new memory allocation is desired. At such an occasion, we transfer some memory from each VM to its respective hypervisor cache and we then perform transparent data access tracing at the hypervisor. With collected traces and derived miss ratio curve for each VM, we determine a VM memory allocation toward our goal. We then release the memory from victim VM's hypervisor cache to the beneficiary VM. We finally release memory from all the hypervisor caches back to respective VMs. Figure 6 illustrates this process.

Our scheme requires a mechanism for dynamic memory adjustment between the VM memory and hypervisor cache. The ballooning technique [23] can serve this purpose. The balloon driver squeezes memory out of a VM by pinning down some memory so the VM cannot use it. The memory can be released back by popping the balloon (*i.e.*, un-pinning the memory).

The size of the hypervisor cache depends on our need of VM page miss rate information. Specifically, more complete page miss rate information (starting from a smaller candidate memory size) demands a smaller VM direct memory (and thus a larger hypervisor cache). When the predicted page miss rate information is used to guide VM memory allocation, our desired completeness of such information depends on the adjustment threshold (the maximum amount of memory we are willing to take away from the VM).

For the purpose of acquiring VM data access pattern and guiding memory allocation, the hypervisor cache can release memory back to the VM as soon as an appropriate VM memory allocation is determined. Since a new memory allocation is typically only needed once in a while, the management overhead of the hypervisor cache is amortized over a long period of time. However, if the management overhead of the hypervisor cache is not considered significant, we may keep the hypervisor cache permanently so we can quickly adjust to any new VM data access behaviors.

It should be noted that there is an inherent delay in our allocation scheme reacting to VM memory need changes — it takes a while to collect sufficient I/O access trace for predicting VM page miss ratio curve. As a result, our scheme may not be appropriate for continuously fluctuating memory re-allocations under very dynamic and adaptive VM workloads.

4.4 Vulnerability to VM Manipulation

One security concern with our adaptive memory allocation is that a selfish or malicious VM may exaggerate its memory requirement to acquire more memory allocation than needed. This might appear particularly problematic for our VM memory requirement estimation based on VM-provided page eviction information. More specifically, a selfish or malicious VM may artificially boost page eviction events so that the hypervisor would predict higher-than-actual VM memory requirement. However, we point out that the VM may achieve the same goal of exaggerating its memory requirement by artificially adding unnecessary I/O reads. Although we cannot prevent a VM from exaggerating its memory requirement, its impact on other VMs' performance is limited as long as we adhere to an appropriate performance isolation constraint.

5 Prototype Implementation

We made a proof-of-concept prototype implementation of the proposed hypervisor cache on Xen virtual machine platform (version 3.0.2) [2]. On this platform, the VMs are called xenU domains and the VM OS is a modified Linux 2.6.16 kernel. The hypervisor includes a thin core (called "hypervisor" in Xen) and a xen0 domain which runs another modified Linux 2.6.16 kernel (with more device driver support).

Our change to the xenU OS is small, mostly about notifying the hypervisor for page evictions. Since an evicted page may be transferred into the hypervisor cache, we must ensure that the page is not reused until the hypervisor finishes processing it. We achieve this by implementing the page eviction notification as a new type of I/O request. Similar to a write request, the source page will not be reused until the request returns, indicating the completion of the hypervisor processing.

The hypervisor cache and mapping tables are entirely implemented in the xen0 domain as part of the I/O backend driver. This is to keep the Xen core simple and small. The storage location in our implementation is represented by a triplet (`major device number, minor device number, block address on the device`). The cache is organized in a queue of pages. Both the cache and mapping tables are indexed with hash tables to speed up the lookup. Our hypervisor cache supports both delayed writes and write-through.

The main purpose of our prototype implementation is to demonstrate the correctness of our design and to illustrate the effectiveness of its intended utilization. At the time of this writing, our implementation is not yet fully optimized. In particular, we use explicit page copying when transferring data between the VM memory and the hypervisor cache. A page remapping technique is used in Xen to pass incoming network packet from the privileged driver domain (xen0) to a normal VM (xenU). However, our measured cost for this page remapping technique does not exhibit significant advantage compared to the explicit page copying, which is also reported in an earlier study [11].

Primitive operations	*Overhead*
Mapping table lookup	0.28 μs
Mapping table insert	0.06 μs
Mapping table delete	0.06 μs
Cache lookup	0.28 μs
Cache insert (excl. page copying)	0.13 μs
Cache delete	0.06 μs
Cache move to tail	0.05 μs
Page copying	7.82 μs

Table 1: Overhead of primitive cache management operations on a Xeon 2.0 GHz processor.

Table 1 lists the overhead of primitive cache management operations on a Xeon 2.0 GHz processor. Each higher-level function (read cache hit, read cache miss, write cache hit, write cache miss, eviction cache hit, and eviction cache miss) is simply the combination of several primitive operations. Page copying is the dominant cost for read cache hit and eviction cache miss. The cost for other functions is within 1 μs.

6 Evaluation

We perform experimental evaluation on our prototype hypervisor cache. The purpose of our experiments is to validate the correctness of our cache design and implementation (Section 6.2), evaluate its performance and management overhead (Section 6.3), validate its VM miss ratio curve prediction (Section 6.4), and demonstrate its effectiveness in supporting multi-VM memory allocation with flexible QoS objectives (Section 6.5). The experimental platform consists of machines each with one 2.0 GHz Intel Xeon processor, 2 GB of physical memory, and two IBM 10 KRPM SCSI drives.

6.1 Evaluation Workloads

Our evaluation workloads include a set of microbenchmarks and realistic applications/benchmarks with significant data accesses. All workloads are in the style of on-demand services.

Microbenchmarks allow us to examine system behaviors for services of specifically chosen data access patterns. All microbenchmarks we use access a dataset of 500 4 MB disk-resident files. On the arrival of each request, the service daemon spawns a thread to process it. We employ four microbenchmarks with different data access patterns:

- *Sequential*: We sequentially scan through all files one by one. We repeat the sequential scan after all files are accessed. Under LRU page replacement, this access pattern should result in no cache hit when the memory size is smaller than the total data size.
- *Random*: We access files randomly with uniform randomness — each file has an equal probability of being chosen for each access.
- *Zipf*: We access files randomly with a Zipf distribution — file i ($1 \leq i \leq 500$) is accessed with a probability proportional to $\frac{1}{i^{\alpha}}$. The exponent $\alpha = 1.0$ in our test.
- *Class*: We divide files into two classes: one tenth of all files are in the popular class and the rest are in the normal class. Each file in the popular class is 10 times more likely to be accessed than each file in the normal class.

Each microbenchmark also has an adjustable write ratio. The write ratio indicates the probability for each file access to be a write. A read file access reads the entire file content in 64 KB chunks. A write file access overwrites the file with new content of the same size. Writes are also performed in 64 KB chunks.

In addition to the microbenchmarks, our experiments also include four realistic data-intensive services.

- *SPECweb99:* We implemented the static content portion of the SPECweb99 benchmark [19] using the Apache 2.0.44 Web server. This workload contains 4 classes of files with sizes at 1 KB, 10 KB, 100 KB, and 1,000 KB respectively and the total dataset size is 4.9 GB. During each run, the four classes of files are accessed according to a distribution that favors small files. Within each class, a Zipf distribution with exponent $\alpha = 1.0$ is used to access individual files.
- *Index searching:* We acquired a prototype of the index searching server and a dataset from the Web search engine Ask Jeeves [1]. The dataset contains the search index for about 400,000 Web pages. It includes a 66 MB mapping file that maps MD5-encoded keywords to proper locations in the search index of 2.4 GB. For each keyword in an input query, a binary search is first performed on the mapping file and then search indexes of query keywords are then accessed. The search query words in our test workload are based on a one-week trace recorded at the Ask Jeeves site in 2002.
- *TPC-C:* We include a local implementation of the TPC-C online transaction processing benchmark [20]. TPC-C simulates a population of terminal operators executing Order-Entry transactions against a database. In our experiments, the TPC-C benchmark runs on the MySQL 5.0.18 database with a database size of 2.6 GB.
- *TPC-H:* We evaluate a local implementation of the TPC-H decision support benchmark [21]. The TPC-H workload consists of 22 complex SQL queries. Some queries require excessive amount of time to finish and they are not appropriate for interactive on-demand services. We choose a subset of 17 queries in our experimentation: Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q9, Q11, Q12, Q13, Q14, Q15, Q17, Q19, Q20, and Q22. In our experiments, the TPC-H benchmark runs on the MySQL 5.0.18 database with a database size of 496 MB.

6.2 Cache Correctness

We augment the microbenchmarks to check the correctness of returned data from the hypervisor cache. We do so by maintaining a distinct signature for each 1 KB block of each file during the file creations and overwrites. Each file read access checks the signatures of returned content, which would fail if the content were incorrectly cached. We tested the four microbenchmarks at three different write ratios (0%, 10%, and 50%) and a variety of VM memory and hypervisor cache sizes. We found no signature checking failures over all test runs.

We also ran tests to check the necessity of our cache correctness support described in Section 3.2. We changed the hypervisor so that it does not capture all page eviction/release or that it does not check the reverse mapping from storage locations to VM memory pages when admitting evicted data. We detect incorrect content for both cases and we traced the problems to the error cases described in Section 3.2.

6.3 Performance and Management Overhead

We evaluate the cache performance and management overhead of our hypervisor exclusive cache. For each workload, we configure the total available memory (combined size of the VM memory and hypervisor cache) to be 512 MB. In the baseline scheme, all memory is directly managed by the VM and there is no hypervisor cache. We then examine the cases when we transfer 12.5%, 25%, 50%, and 75% of the memory (or 64 MB, 128 MB, 256 MB, and 384 MB respectively) to be managed by the hypervisor cache. Note that some setting ("75% memory to cache") may not be typical in practice. Our intention is to consider a wide range of conditions in this evaluation.

We look at three performance and overhead metrics: the overall service request throughput (Figure 7), the I/O overhead per request (Figure 8), and CPU overhead in hypervisor cache management (Figure 9). Here the I/O overhead only counts those page I/Os that reach the real

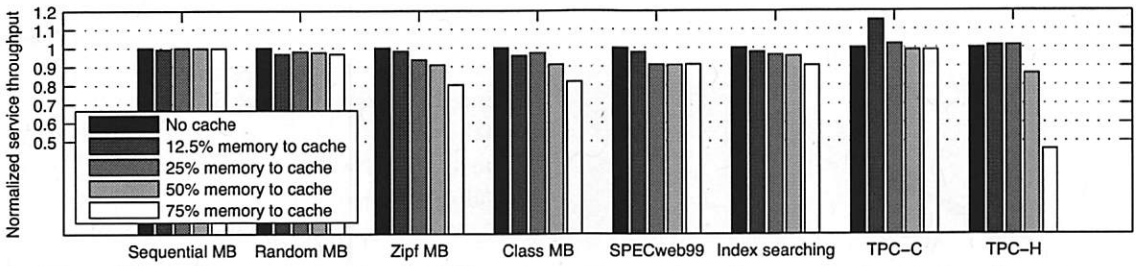


Figure 7: Service request throughput of different hypervisor caching schemes normalized to that of no cache.

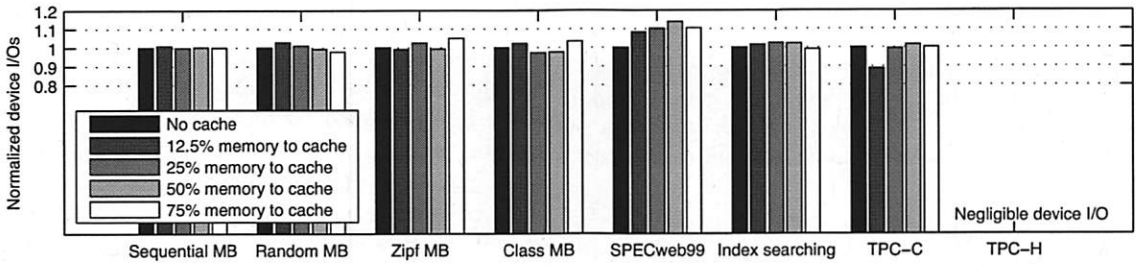


Figure 8: The I/O overhead per request of different schemes normalized to that of no cache.

storage device (*i.e.*, those that miss both VM memory and the hypervisor cache). Note that since we intend to compare the overhead when finishing the same amount of work, the page I/O overhead per request (or workload unit) is a better metric than the page I/O overhead per time unit. In particular, two systems may exhibit the same page I/O overhead per time unit simply because they are both bound by the maximum I/O device throughput. For the same reason, we use a scaled CPU overhead metric. The scaling ratio is the throughput under "no cache" divided by the throughput under the current scheme.

Among the eight workloads in our experimental setup, TPC-H is unique in the sense that it is completely CPU-bound with 512 MB memory. Below we analyze the results separately for it and the other workloads.

Seven non-CPU-bound workloads. In terms of service throughput, the degradation compared to "no cache" is less than 20% in all cases and no more than 9% excluding the extreme condition of "75% memory to cache". This is largely because the employment of our hypervisor cache does not significantly increase the system I/O overhead (as shown in Figure 8). A closer examination discovers an I/O overhead increase of up to 13% for SPECweb99. This is because our hypervisor cache does not cache evicted VM data that is not in OS page buffer, such as file meta-data like `inode` and `dentry` in Linux. Most files accessed by SPECweb99 are very small and thus the effect of not caching file meta-data is more pronounced. Note that the hypervisor caching of file meta-data requires the understanding of file meta-data memory layout, which would severely compromise

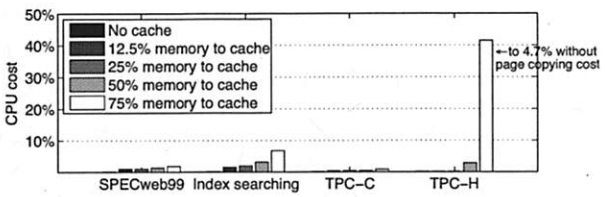


Figure 9: CPU cost of hypervisor cache for four real services. We do not show results for microbenchmarks since they do not contain any realistic CPU workload.

the transparency of the hypervisor cache. Excluding SPECweb99, the page fault rate varies between a 11% decrease and a 5% increase compared to "no cache" over all test cases. Now we consider the CPU overhead incurred by cache management and minor page faults (shown in Figure 9). Overall, the additional overhead (compared to the "no cache" case) is up to 6.7% in all cases and up to 3.2% excluding the extreme condition of "75% memory to cache". Its impact on the performance of non-CPU-bound workloads is not substantial.

CPU-bound TPC-H. There is no real device I/O overhead in all test cases and its performance difference is mainly determined by the amount of additional CPU overhead of the cache management. Such cost is negligible for "12.5% memory to cache" and "25% memory to cache". It is more significant for "50% memory to cache" and "75% memory to cache", causing 14% and 54% throughput degradation respectively compared to "no cache". This is largely due to the costly page copying operations. Excluding the page copying overhead, the expected CPU overhead at "75% memory to cache" would be reduced from 41% to 4.7%. This indicates that

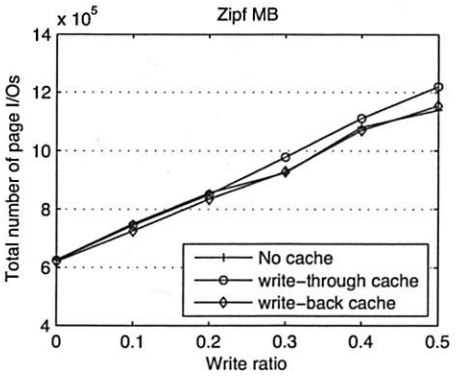


Figure 10: Performance impact of write-through/write-back caches in a system with 256 MB VM direct memory and 256 MB hypervisor cache. The y-axis shows the total number of page I/Os during the Zipf test with 2,000 file requests at different write ratios.

the concern on CPU cost can be significantly alleviated with an efficient page transfer mechanism.

As we discussed in Section 3.1, employing write-through at the hypervisor cache maintains the persistence semantic of the write completion. However, write-through is not as effective as delayed writes in caching the write I/O traffic. To illustrate the performance implication of two write strategies employed by the hypervisor cache, we run the Zipf microbenchmark with different write ratios ranging from 0 (read-only) to 0.5. The result in Figure 10 indicates that write-through indeed yields more I/O operations than the original system (around 7% at 0.5 write ratio), whereas delayed writes does not increase the number of page I/Os.

We summarize the performance and overhead results as follows. The employment of hypervisor cache does not increase the system I/O overhead (excluding an exceptional case). The CPU overhead for our current prototype implementation can be significant, particularly at the extreme setting of "75% memory to cache". However, our results suggest that the CPU overhead does not have large impact on the performance of services that are not CPU-bound. We also expect that a more optimized cache implementation in the future may reduce the CPU cost.

6.4 Accuracy of Miss Ratio Curve Prediction

We perform experiments to validate the accuracy of our VM miss ratio curve (page miss rate *vs.* memory size curve) prediction. Jones *et al.* [12] have demonstrated the prediction of VM miss ratio curve for memory sizes larger than the current allocation. The contribution of our hypervisor cache-based transparent data access tracing is to predict VM miss ratio curve for memory sizes smaller than the current allocation. In practice, we predict a miss ratio curve that includes memory sizes both larger and smaller than the current allocation. Such a curve can tell the performance degradation when the memory allocation is reduced as well as the performance improvement when the memory allocation is increased. Both pieces of information are necessary to determine the VM memory allocation with performance assurance.

We use a system configuration with a large hypervisor cache to produce the VM miss ratio curve over a wide range. With a memory allocation of 512 MB, 384 MB is managed as the hypervisor cache and the VM memory has 128 MB left. This setting allows us to predict the VM miss ratio curve from the memory size of 128 MB. Smaller hypervisor caches may be employed in practice if we have a bound on the maximum amount of VM memory reduction, or if the management overhead for a large cache is considered too excessive. We validate the prediction accuracy by comparing against measured miss ratios at several chosen memory sizes. The validation measurements are performed on VM-only systems with no hypervisor cache.

Figure 11 illustrates the prediction accuracy for the eight workloads over memory sizes between 128 MB and 1024 MB. Results suggest that our prediction error is less than 15% in all validation cases. Further, the error is less than 9% for memory sizes smaller than the current allocation (512 MB), which is the primary target of our hypervisor cache-based miss ratio prediction. We believe the prediction error is due to the imperfect LRU replacement employed in the VM OS.

Since we know the microbenchmark data access patterns, we can also validate their miss ratio curves with simple analysis. Sequential MB has a flat curve since there can be no memory hit as long as the memory size is less than the total data size. Random MB's data access miss rate should be $1 - \frac{\text{memory size}}{\text{data size}}$ and therefore its miss ratio curve is linear. Zipf MB and Class MB have more skewed data access patterns than Random MB so the slopes of their miss ratio curves are steeper.

6.5 Multi-VM Memory Allocation

Guided by the predicted VM miss ratio curves, we perform experiments on multi-VM memory allocation with performance assurance. Our experiments are conducted with the allocation goals of *isolated sharing* and *VMware ESX server emulation* described in Section 4.2 respectively. In our experiments, we employ three VMs, running SPECweb99, index searching, and TPC-H respectively. The initial baseline memory allocation for each VM is 512 MB. We adjust the total 1,536 MB memory among the three VMs toward our allocation goal.

Isolated sharing In isolated sharing experiments, we attempt to minimize a system-wide page miss metric

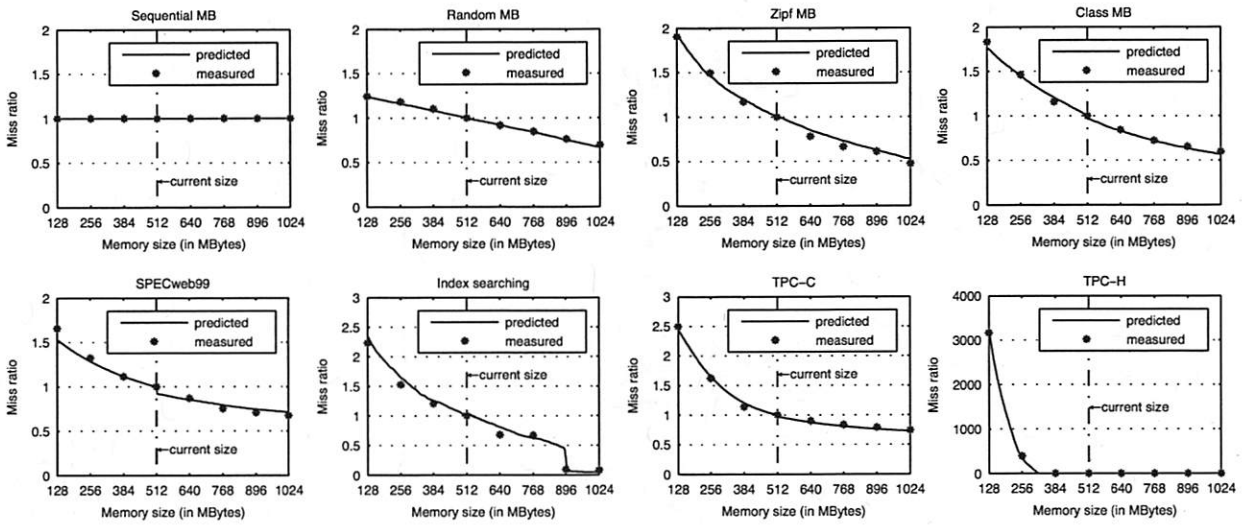


Figure 11: Accuracy of miss ratio curve prediction for memory sizes between 128 MB and 1024 MB. The current memory allocation is 512 MB, within which 384 MB is managed by the hypervisor cache. The miss ratio at each memory size is defined as the number of page misses at the current memory size divided by the page miss number at a baseline memory size (512 MB in this case). Since there is almost no page miss for TPC-H at 512 MB memory, we set a small baseline page miss number for this workload to avoid numeric imprecision in divisions.

(*e.g.*, profitable sharing) while at the same time no VM should experience a performance loss beyond a given bound (*i.e.*, isolation constraint). In our experiments, the system-wide page miss metric is the geometric mean of all VMs' miss ratios which represents the average acceleration ratio for all VMs in terms of page miss reduction. The performance loss bound is set as a maximum percentage increase of page misses compared to the baseline allocation. To demonstrate the flexibility of our policy, we run two experiments with different isolation constraints — 5% and 25% performance loss bounds respectively. We allocate memory in the multiple of 4 MB. We use exhaustive search to find the optimal allocation strategy and the computation overhead for the exhaustive search is acceptable for three VMs.

Table 2 lists the memory allocation results. Overall, the experimental results show that our hypervisor cache-based memory allocation scheme can substantially reduce the system-wide page miss metric (15% average page miss reduction at 5% isolation constraint and 59% average page miss reduction at 25% isolation constraint). This is primarily due to our ability of transparent data access tracing and accurate VM miss ratio curve prediction. The two very different allocation outcomes at different isolation constraints demonstrate the flexibility and performance assurance of our approach. In comparison, a simple working set-based allocation approach [23] may not provide such support.

Generally all VMs observe the isolation constraints in our experiments. However, a small violation is observed for SPECweb99 in the test with 25% isolation constraint

Initial configuration

	VM #1	VM #2	VM #3
Workload	SPECweb	Searching	TPC-H
Memory alloc.	512 MB	512 MB	512 MB

Allocation with 5% isolation constraint

	VM #1	VM #2	VM #3
Memory alloc.	452 MB	748 MB	336 MB
Predicted miss ratio	1.05	0.64	1.00
Predicted geo. mean	0.88		
Measured miss ratio	1.04	0.58	1.00
Measured geo. mean	0.85		

Allocation with 25% isolation constraint

	VM #1	VM #2	VM #3
Memory alloc.	280 MB	920 MB	336 MB
Predicted miss ratio	1.24	0.06	1.00
Predicted geo. mean	0.43		
Measured miss ratio	1.28	0.05	1.00
Measured geo. mean	0.41		

Table 2: Memory allocation results for isolated sharing.

(28% page miss increase). This is due to the miss ratio curve prediction error. We believe such a small violation is tolerable in practice. If not, we can leave an error margin when determining the allocation (*e.g.*, using a 20% isolation constraint on the predicted miss ratios when a hard 25% isolation constraint needs to be satisfied).

VMware ESX server emulation This experiment demonstrates that hypervisor cache-based allocation is

Initial configuration

	VM #1	VM #2	VM #3
Workload	SPECweb	Searching	TPC-H
Memory alloc.	512 MB	512 MB	512 MB

Hcache emulation

	VM #1	VM #2	VM #3
Memory alloc.	600 MB	600 MB	336 MB
Measured miss ratio	0.85	0.71	1.00

VMware ESX server

	VM #1	VM #2	VM #3
Memory alloc.	576 MB	576 MB	384 MB
Measured miss ratio	0.88	0.78	1.00

Hcache emulation (A background task runs with TPC-H)

	VM #1	VM #2	VM #3
Memory alloc.	578 MB	578 MB	380 MB
Measured miss ratio	0.88	0.78	1.00

VMware ESX server (A background task runs with TPC-H)

	VM #1	VM #2	VM #3
Memory alloc.	512 MB	512 MB	512 MB
Measured miss ratio	1.00	1.00	1.00

Table 3: Memory allocation results for VMware ESX server emulation.

able to: 1) emulate the memory allocation policy employed in VMware ESX server; and 2) more accurately discover VM memory need when choosing victim VMs. We employ VMware ESX server version 3.0.0 in this test. We ported our test workloads to ESX server environment and conditioned all workload parameters in the same way. For each VM, the initial and maximum memory allocations are 512 MB and 1 GB respectively. All VMs receive the same share (a VMware ESX server parameter indicating a per-VM proportional right to memory).

We first use the exactly same three VMs as in the isolated sharing experiments. Table 3 shows that both ESX server and our emulation are able to reclaim unused memory from TPC-H VM without raising the page fault rate. However, ESX server is more conservative, resulting in less performance improvement for other VMs. We suspect this conservatism is related to the prediction inaccuracy inherent with its memory sampling approach.

Then we add a light background workload to TPC-H VM. The workload touches one 4 MB file every 2.5 seconds over 500 such files repeatedly. Figure 12 shows the predicted and measured miss ratio curve for this new TPC-H VM. It is clear that beyond the allocation of around 360 MB, more memory does not reduce the VM page fault rate. Our hypervisor cache-based allocation

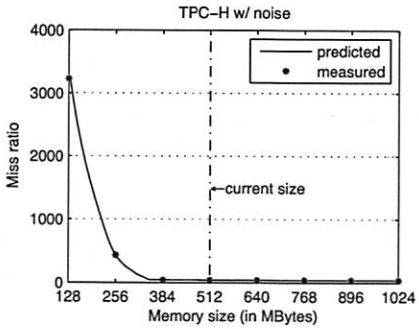


Figure 12: Miss ratio curve of TPC-H with a light background task.

correctly recognizes this and therefore takes away some TPC-H VM memory to the other two VMs. In contrast, ESX server estimates that the TPC-H VM has a large working set (around 830 MB) and thus does not perform any memory allocation adjustment. This is due to inherent weakness of the working set model-based memory allocation [6, 18].

7 Conclusion

For data-intensive services on a virtual machine (VM) platform, the knowledge of VM page misses under different memory resource provisioning is desirable for determining appropriate VM memory allocation and for facilitating service consolidation. In this paper, we demonstrate that the employment of a hypervisor-level exclusive buffer cache can allow transparent data access tracing and accurate prediction of the VM page miss ratio curve without incurring significant overhead (no I/O overhead and mostly small CPU cost). To achieve this goal, we propose the design of the hypervisor exclusive cache and address challenges in guaranteeing the cache content correctness when the data enters the cache directly from the VM memory.

As far as we know, existing hypervisor-level buffer cache is used primarily for the purpose of keeping single copy of data shared across multiple VMs. Our hypervisor exclusive cache is unique in its ability to manage large chunk of a VM's memory without increasing the overall system page faults. Although our utilization of this cache is limited to transparent data access tracing in this paper, there might also be other beneficial use of the cache. For example, the hypervisor-level buffer cache allows the employment of new cache replacement policy and I/O prefetching policy transparent to the VM OS. This may be desirable when the OS-level caching and I/O prefetching are not fully functional (*e.g.*, during OS installation or boot [10]) or when the default OS-level policy is insufficient (*e.g.*, desiring more aggressive I/O prefetching [14]).

Acknowledgments We would like to thank Irfan Ahmad (VMware Inc.) for helpful discussions in the preparation of this work. We are also grateful to the USENIX anonymous reviewers for their useful comments that improved this paper.

References

[1] Ask jeeves search. http://www.ask.com.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, October 2003.

[3] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Trans. on Computer Systems*, 15(4):412–447, November 1997.

[4] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The collective: A cache-based system management architecture. In *Proc. of the Second USENIX Symp. on Networked Systems Design and Implementation*, pages 259–272, Boston, MA, May 2005.

[5] Z. Chen, Y. Zhou, and K. Li. Eviction based placement for storage caches. In *Proc. of the USENIX Annual Technical Conf.*, pages 269–282, San Antonio, TX, June 2003.

[6] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational databases systems. In *Proc. of the 11th Int'l Conf. on Very Large Data Bases*, pages 127–141, Stockholm, Sweden, August 1985.

[7] C. Clark, K. Fraser, S. Hand, J. J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of the Second USENIX Symp. on Networked Systems Design and Implementation*, pages 273–286, Boston, MA, May 2005.

[8] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. US Patent, 6397242, October 1998.

[9] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, pages 154–169, Kiawah Island, SC, December 1999.

[10] I. Ahmad, VMware Inc., July 2006. Personal communication.

[11] Y. Turner J. R. Santos, G. J. Janakiraman. Network optimizations for PV guests. Xen Summit, September 2006. http://www.xensource.com/files/summit_3/networkoptimizations.pdf.

[12] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 13–23, San Jose, CA, October 2006.

[13] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[14] C. Li, K. Shen, and A. E. Papathanasiou. Competitive prefetching for concurrent sequential I/O. In *Proc. of the Second EuroSys Conf.*, pages 189–202, Lisbon, Portugal, March 2007.

[15] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[16] M. K. McKusick. Disks from the perspective of a file system. *USENIX ;login:*, 31(3):18–19, June 2006.

[17] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 79–95, Copper Mountain Resort, CO, December 1995.

[18] G. M. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Trans. on Database Systems*, 11(4):473–498, December 1986.

[19] SPECweb99 benchmark. http://www.specbench.org/osg/web99.

[20] Transaction Processing Performance Council. TPC benchmark C. http://www.tpc.org/tpcc.

[21] Transaction Processing Performance Council. TPC benchmark H. http://www.tpc.org/tpch.

[22] VMware Infrastructure - ESX Server. http://www.vmware.com/products/esx.

[23] C. A. Waldspurger. Memory resource management in vmware esx server. In *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation*, pages 181–194, Boston, MA, December 2002.

[24] M. Williamson. Xen wiki: XenFS. http://wiki.xensource.com/xenwiki/XenFS.

[25] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proc. of the USENIX Annual Technical Conf.*, pages 161–175, Monterey, CA, June 2002.

[26] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation*, pages 103–116, Seattle, WA, November 2006.

[27] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proc. of the 11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 177–188, Boston, MA, October 2004.

Hyperion: High Volume Stream Archival for Retrospective Querying

Peter J. Desnoyers
Department of Computer Science
University of Massachusetts

Prashant Shenoy
Department of Computer Science
University of Massachusetts

Abstract

Network monitoring systems that support data archiving and after-the-fact (retrospective) queries are useful for a multitude of purposes, such as anomaly detection and network and security forensics. Data archiving for such systems, however, is complicated by (a) data arrival rates, which may be hundreds of thousands of packets per second on a single link, and (b) the need for online indexing of this data to support retrospective queries. At these data rates, both common database index structures and general-purpose file systems perform poorly.

This paper describes Hyperion, *a system for archiving, indexing, and on-line retrieval of high-volume data streams. We employ a write-optimized* stream file system *for high-speed storage of simultaneous data streams, and a novel use of* signature file *indexes in a distributed multi-level index.*

We implement Hyperion on commodity hardware and conduct a detailed evaluation using synthetic data and real network traces. Our streaming file system, StreamFS, is shown to be fast enough to archive traces at over a million packets per second. The index allows queries over hours of data to complete in as little as 10-20 seconds, and the entire system is able to index and archive over 200,000 packets/sec while processing simultaneous on-line queries.

1 Introduction

Motivation: Network monitoring by collecting and examining packet headers has become popular for a multitude of management and forensic purposes, from tracking the perpetrators of system attacks to locating errors or performance problems. Networking monitoring systems come in two flavors. In *live* monitoring, packets are captured and examined in real-time by the monitoring system. Such systems can run continual queries on the packet stream to detect specific conditions [20], compute and continually update traffic statistics, and proactively detect security attacks by looking for worm or denial of service signatures [9]. Regardless of the particular use, in live monitoring systems, captured packet headers and payloads are discarded once examined.

However, there are many scenarios where it is useful to retain packet headers for a period of time. Network forensics is one such example—the ability to "go back" and retrospectively examine network packet headers is immensely useful for network troubleshooting (e.g., root-cause analysis), to determine how an intruder broke into a computer system, or to determine how a worm entered a particular administrative domain. Such network monitoring systems require *archival storage* capabilities, in addition to the ability to query and examine live data. Besides capturing data at wire speeds, these systems also need to archive and index data at the same rates. Further, they need to efficiently retrieve archived data to answer *retrospective* queries.

Currently, there are two possible choices for architecting an archiving system for data streams. A relational database may be used to archive data, or a custom index may be created on top of a conventional file system.

The structure of captured information—a header for each packet consisting of a set of fields—naturally lends itself to a database view. This has led to systems such as GigaScope [6] and MIND [18], which implement a SQL interface for querying network monitoring data.

A monitoring system must receive new data at high rates: a single gigabit link can generate hundreds of thousands of packet headers per second and tens of Mbyte/s of data to archive, and a single monitoring system may record from multiple links. These rates have prevented the use of traditional database systems. MIND, which is based on a peer-to-peer index, extracts and stores only flow-level information, rather than raw packet headers. GigaScope is a stream database, and like other stream databases to date [20, 27, 1] supports continual queries on live streaming data; data archiving is not a design concern in these systems. GigaScope, for instance, can process continual queries on data from some of the highest-speed links in the Internet, but relies on external mechanisms to store results for later reference.

An alternative is to employ a general-purpose file sys-

tem to store captured packet headers, typically as log files, and to construct a special-purpose index on these files to support efficient querying. A general-purpose file system, however, is not designed to exploit the particular characteristics of network monitoring applications, resulting in lower system throughput than may be feasible. Unix-like file systems, for instance, are typically optimized for writing small files and reading large ones sequentially, while network monitoring and querying writes very large files at high data rates, while issuing small random reads. Due to the high data volume in these applications, and the need to bound worst-case performance in order to avoid data loss, it may be desirable to optimize the system for these access patterns instead of relying on a general-purpose file system.

Thus, the unique demands placed by high-volume stream archiving indicate that neither existing databases nor file systems are directly suited to handle their storage needs. This motivates the need for a new storage system that runs on commodity hardware and is specifically designed to handle the needs of high-volume stream archiving in the areas of disk performance, indexing, data aging, and query and index distribution.

Research Contributions: In this paper, we present *Hyperion*[1], a novel stream archiving system that is designed for storing and indexing high-volume packet header streams. Hyperion consists of three components: (i) *StreamFS*, a stream file system that is optimized for sequential immutable streaming writes, (ii) a multi-level index based on signature files, used in the past by text search engines, to sustain high update rates, and (iii) a distributed index layer which distributes coarse-grain summaries of locally archived data to other nodes, to enable distributed querying.

We have implemented Hyperion on commodity Linux servers, and have used our prototype to conduct a detailed experimental evaluation using real network traces. In our experiments, the worst-case StreamFS throughput for streaming writes is 80% of the mean disk speed, or almost 50% higher than for the best general-purpose Linux file system. In addition, StreamFS is shown to be able to handle a workload equivalent to streaming a million packet headers per second to disk while responding to simulateneous read requests. Our multi-level index, in turn, scales to data rates of over 200K packets/sec while at the same time providing interactive query responses, searching an hour of trace data in seconds. Finally, we examine the overhead of scaling a Hyperion system to tens of monitors, and demonstrate the benefits of our distributed storage system using a real-world example.

The rest of this paper is structured as follows. Section 2 and 3 present design challenges and guiding design principles. Sections 4, 5, and 6 present the design and implementation of Hyperion. We present experimental results in Section 7, related work in Section 8, and our conclusions in Section 9.

2 Design Challenges

The design of a high-volume archiving and indexing system for data streams must address several challenges:

Archive multiple, high-volume streams. A single heavily loaded gigabit link may easily produce monitor data at a rate of 20Mbyte/sec[2]; a single system may need to monitor several such links, and thus scale far beyond this rate. Merely storing this data as it arrives may be a problem, as a commodity hardware-based system of this scale must necessarily be based on disk storage; although the peak speed of such a system is sufficient, the worst-case speed is far lower than is required. In order to achieve the needed speeds, it is necessary to exploit the characteristics of modern disks and disk arrays as well as the sequential append-only nature of archival writes.

Maintain indices on archived data — The cost of exhaustive searches through archived data would be prohibitive, so an index is required to support most queries. Over time, updates to this index must be stored at wireline speed, as packets are captured and archived, and thus must support especially efficient updating. This high update rate (e.g. 220K pkts/sec in the example above) rules out many index structures; e.g. a B-tree index over the entire stream would require one or more disk operations per insertion. Unlike storage performance requirements, which must be met to avoid data loss, retrieval performance is not as critical; however, our goal is that it be efficient enough for interactive use. The target for a highly selective query, returning very few data records, is that it be able to search an hour of indexed data in 10 seconds.

Reclaim and re-use storage. Storage space is limited in comparison to arriving data, which is effectively infinite if the system runs long enough. This calls for a mechanism for reclaiming and reusing storage. Data aging policies that delete the oldest or the least-valuable data to free up space for new data are needed, and data must be removed from the index as it is aged out.

Coordinate between monitors. A typical monitoring system will comprise multiple monitoring nodes, each monitoring one or more network links. In network forensics, for instance, it is sometime necessary to query data archived at multiple nodes to trace events (e.g. a worm) as they move through a network. Such distributed querying requires some form of coordination between monitoring nodes, which involves a trade-off between distribution of data and queries. If too much data or index information is distributed across monitoring nodes, it may

[1] Hyperion, a Titan, is the Greek god of observation.

[2] 800Mbit/sec traffic, 450 byte packets, 90 bytes captured per packet

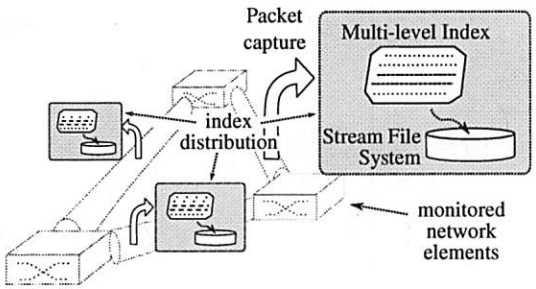


Figure 1: Components of the Hyperion network monitoring system.

limit the overall scale of the system as the number of nodes increase; if queries must be flooded to all monitors, query performance will not scale.

Run on *commodity hardware.* The use of commodity processors and storage imposes limits on the processing and storage bandwidths available at each monitoring node, and the system must optimize its resource usage to scale to high data volumes.

3 Hyperion Design Principles

The challenges outlined in the previous section result in three guiding principles for our system design.

P1: *Support queries, not reads:* A general-purpose file system supports low-level operations such as reads and writes. However, the nature of monitoring applications dictates that data is typically accessed in the form of queries; in the case of Hyperion, for instance, these queries would be predicates identifying values for particular packet header fields such as source and destination address. Consequently, a stream archiving system should support data accesses at the level of queries, as opposed to raw reads on unstructured data. Efficient support for querying implies the need to maintain an index and one that is particularly suited for high update rates.

P2: *Exploit sequential, immutable writes:* Stream archiving results in continuous sequential writes to the underlying storage system; writes are typically immutable since data is not modified once archived. The system should employ data placement techniques that exploit these I/O characteristics to reduce disk seek overheads and improve system throughput.

P3: *Archive locally, summarize globally.* There is an inherent conflict between the need to scale, which favors local archiving and indexing to avoid network writes, and the need to avoid flooding to answer distributed queries, which favors sharing information across nodes. We "resolve" this conflict by advocating a design where data archiving and indexing is performed locally and a coarse-grain summary of the index is shared between nodes to support distributed querying without flooding.

Based on these principles, we have designed *Hyperion*, a stream archiving system that consists of three key components: (i) a *stream file system* that is highly optimized for high volume archiving and retrospective querying, (ii) a *multi-level index structure* that is designed for high update rates while retaining reasonable lookup performance, and (iii) a *distributed index* layer that distributes a coarse-grain summary of the local indices to enable distributed queries (see Figure 1) The following sections present the rationale for and design of these components in detail.

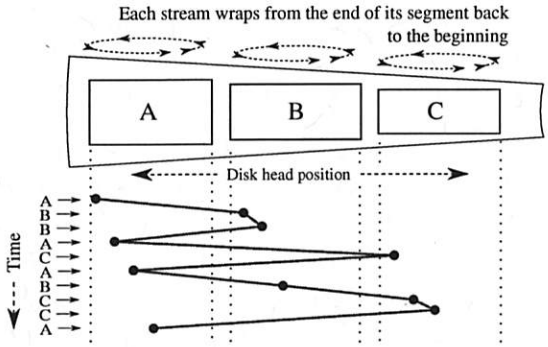


Figure 2: Write arrivals and disk accesses for single file per stream. Writes for streams A, B, and C are interleaved, causing most operations to be non-sequential.

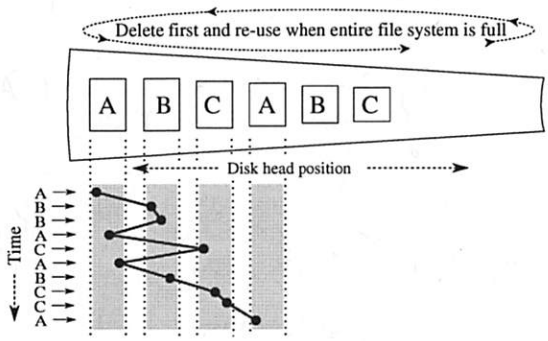


Figure 3: Logfile rotation. Data arrives for streams A, B, and C in an interleaved fashion, but is written to disk in a mostly sequential order.

4 Hyperion Stream File System

The requirements for the Hyperion storage system are: storage of multiple high-speed traffic streams without loss, re-use of storage on a full disk, and support for concurrent read activity without loss of write performance. The main barrier to meeting these requirements is the variability in performance of commodity disk and array storage; although storage systems with best-case throughput sufficient for this task are easily built, worst-case throughput can be three orders of magnitude worse.

In this section we first consider implementing this storage system on top of a general-purpose file system. After

exploring the performance of several different conventional file systems on stream writes as generated by our application, we then describe StreamFS, an application-specific file system for stream storage.[3]

In order to consider these issues, we first define a stream storage system in more detail. Unlike a general purpose file system which stores *files*, a stream storage system stores *streams*. These streams are:

- *Recycled:* when the storage system is full, writes of new data succeed, and old data is lost (i.e. removed or overwritten in a circular buffer fashion). This is in contrast to a general-purpose file system, where new data is lost and old data is retained.
- *Immutable:* an application may append data to a stream, but does not modify previously written data.
- *Record-oriented:* attributes such as timestamps are associated with ranges in a stream, rather than the stream itself. Optionally, as in StreamFS, data may be written in records corresponding to these with boundaries which are preserved on retrieval.

This stream abstraction provides the features needed by Hyperion, while lacking other features (e.g. mutability) which are un-needed for our purposes.

4.1 Why Not a General Purpose Filesystem?

To store streams such as this on a general purpose file system, a mapping between streams and files is needed. A number of such mappings exist; we examine several of them below. In this consideration we ignore the use of buffering and RAID, which may be used to improve the performance of each of these methods but will not change their relative efficiency.

File-per-stream: A naïve stream storage implementation may be done by creating a single large file for each data stream. When storage is filled, the beginning of the file cannot be deleted if the most recent data (at the end of the file) is to be retained, so the beginning of the file is over-written with new data in circular buffer fashion. A simplified view of this implementation and the resulting access patterns may be seen in Figure 2. Performance of this method is poor, as with multiple simultaneous streams the disk head must seek back and forth between the write position on each file.

Log files: A better approach to storing streams is known as *logfile rotation*, where a new file is written until it reaches some maximum size, and then closed; the oldest files are then deleted to make room for new ones. Simplified operation may be seen in Figure 3, where files

[3]Specialized file systems for application classes (e.g. streaming media) have a poor history of acceptance. However, file systems specific to a single application, often implemented in user space, have in fact been used with success in a number of areas such as web proxies [25] and commercial databases such as Oracle. [21]

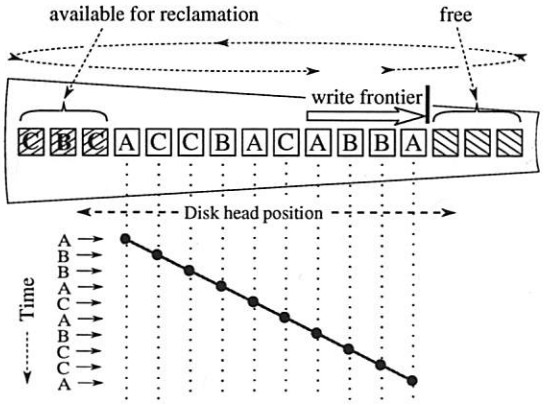


Figure 4: Log allocation - StreamFS, LFS. Data arrives in an interleaved fashion and is written to disk in that same order.

are allocated as single extents across the disk. This organization is much better at allocating storage flexibly, as allocation decisions may be revised dynamically when choosing which file to delete. As shown in the figure, fairly good locality will be maintained when first filling the volume; with continued use, however, consecutively created files and extents may be located far apart on disk, degrading throughput to that of the previous method.

Log-Structured File System: The highest write throughput will be obtained if storage is allocated sequentially as data arrives, as illustrated in Figure 4. This is the method used by Log-structured File Systems (LFS) such as [22], and when logfile rotation is used on such a file system, interleaved writes to multiple streams will be allocated closely together on disk.

Although write allocation in log-structured file systems is straightforward, *cleaning*, or the garbage collecting of storage space after files are deleted, has however remained problematic [24, 32]. Cleaning in a general-purpose LFS must handle files of vastly different sizes and lifetimes, and all existing solutions involve copying data to avoid fragmentation. The FIFO-like Hyperion write sequence is a very poor fit for such general cleaning algorithms; in Section 7 our results indicate that it results in significant cleaning overhead.

4.2 StreamFS Storage Organization

The Hyperion stream file system, StreamFS, adopts the log structured write allocation of LFS; as seen in Figure 4, all writes take place at the *write frontier*, which advances as data is written. LFS requires a garbage collector, the *segment cleaner* to eliminate fragmentation which occurs as files are deleted; however, StreamFS does not require this, and *never copies data in normal operation.* This eliminates the primary drawback of log-structured file systems and is made possible by taking advantage of both the StreamFS storage reservation sys-

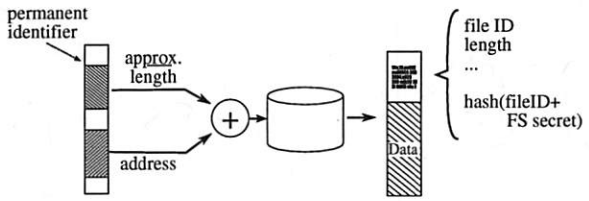


Figure 5: Random read operation.

tem and the properties of stream data.

The trivial way to do this would be to over-write all data as the write frontier advances, implicitly establishing a single age-based expiration policy for all streams. Such a policy would not address differences between streams in both rate and required data retention duration. Instead, StreamFS provides a storage *guarantee* to each stream; no records from a stream will be reclaimed or over-written while the stream size (i.e. retained records) is less than this guarantee. Conversely, if the size of a stream is larger than its guarantee, then only that amount of recent data is protected, and any older records are considered *surplus*.

The sum of guarantees is constrained to be less than the size of the storage system minus a fraction; we term the ratio of guarantees to volume size the volume *utilization*.[4] As with other file systems the utilization has a strong effect on performance.

StreamFS avoids a segment cleaner by writing data in small fixed-length blocks (default 1MB); each block stores data from a single stream. As the write frontier advances, it is only necessary to determine whether the next block is surplus. If so, it is simply overwritten, as seen in Figure 4, and if not it is skipped and will expire later; no data copying or cleaning is needed in either case. This provides a flexible storage allocation mechanism, allowing storage reservation as well as best-effort use of remaining storage. Simulation results have shown [8] this "cleaning" strategy to perform very well, with no virtually no throughput degradation for utilizations of 70% or less, and no more than a 15% loss in throughput at 90% utilization.

4.3 Read Addressing via Persistent Handles

Hyperion uses StreamFS to store packet data and indexes to that data, and then handles queries by searching those indexes and retrieving matching data. This necessitates a mechanism to identify a location in a data stream by some sort of pointer or *persistent handle* which may be stored in an index (e.g. across system restart), and then later used to retrieve the corresponding data. This value could be a byte offset from the start of the stream, with appropriate provisions (such as a 64-bit length) to guard against wrap-around. However, the pattern of reads generated by the index is highly non-sequential, and thus translating an offset into a disk location may require multiple accesses to on-disk tables. We therefore use a mechanism similar to a SYN cookie [2], where the information needed to retrieve a record (i.e. disk location and approximate length) is safely encoded and given to the application as a handle, providing both a persistent handle and a highly optimized random read mechanism.

[4]This definition varies slightly from that used for general-purpose file systems, as much of the "free" space beyond the volume utilization may hold accessible data.

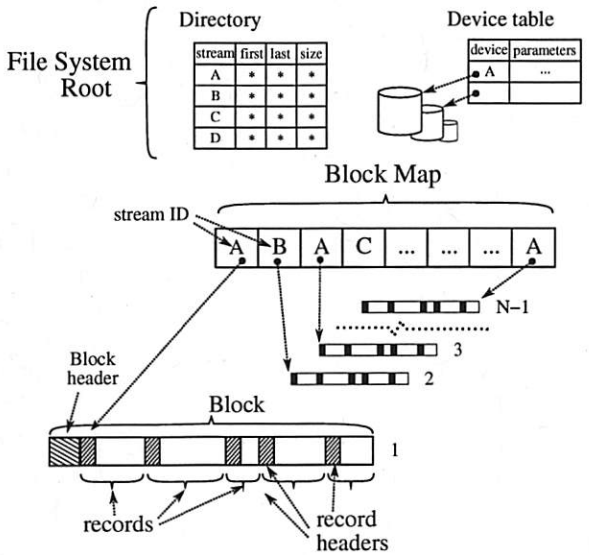


Figure 6: StreamFS metadata structures: record header for each record written by the application, block header for each fixed-length block, block map for every N (256) blocks, and one file system root.

Using application-provided information to directly access the disk raises issues of robustness and security. Although we may ignore security concerns in a single-application system, we still wish to ensure that in any case where a corrupted handle is passed to StreamFS, an error is flagged and no invalid data is returned. This is done by using a *self-certifying* record header, which guarantees that a handle is valid and that access is permitted. This header contains the ID of the stream to which it belongs and the permissions of that stream, the record length, and a hash of the header fields (and a *file system secret* if security is of concern) allowing invalid or forged handles to be detected. To retrieve a record by its persistent handle, StreamFS decodes the handle, applies some simple sanity checks, reads from the indicated address and length, and then verifies the record header hash. At this point a valid reader has been found; permission fields may then be checked and the record returned to the application if appropriate.

4.4 StreamFS Organization

The record header used for self-certifying reads is one of the StreamFS on-disk data structures illustrated in Figure 6. These structures and their fields and functions are as follows:

- *record:* Each variable-length record written by the application corresponds to an on-disk record and *record header*. The header contains validation fields described above, as well as timestamp and length fields.
- *block:* Multiple records from the same stream are combined in a single fixed-length block, by default 1Mbyte in length. The *block header* identifies the stream to which the block belongs, and record boundaries within the block.
- *block map:* Every N^{th} block (default 256) is used as a *block map*, indicating the associated stream and an in-stream sequence number for each of the preceding $N-1$ blocks. This map is used for write allocation, when it must be determined whether a block is part of a stream's guaranteed allocation and must be skipped, or whether it may be overwritten.
- *file system root:* The root holds the stream directory, metadata for each stream (head and tail pointers, size, parameters), and a description of the devices making up the file system.

4.5 Striping and Speed Balancing

Striping: StreamFS supports multiple devices directly; data is distributed across the devices in units of a single block, much as data is striped across a RAID-0 volume. The benefits of single disk write-optimizations in StreamFS extend to multi-disk systems as well. Since successive blocks (e.g., block i and $i+1$) map onto successive disks in a striped system, StreamFS can extract the benefits of I/O parallelism and increase overall system throughput. Further, in a d disk system, blocks i and $i+d$ will map to the same disk drive due to wrap-around. Consequently, under heavy load when there are more than d outstanding write requests, writes to the same disk will be written out sequentially, yielding similar benefits of sequential writes as in a single-disk system.

Speed balancing: Modern disk drives are *zoned* in order to maintain constant linear bit density; this results in disk throughput which can differ by a factor of 2 between the innermost and the outermost zones. If StreamFS were to write out data blocks sequentially from the outer to inner zones, then the system throughput would drop by a factor of two when the write frontier reached the inner zones. This worst-case throughput, rather than the mean throughput, would then determine the maximum loss-less data capture rate of the monitoring system.

StreamFS employs a balancing mechanism to ensure that system throughput remains roughly constant over time, despite variations across the disk platter. This is done by appropriately spreading the write traffic across the disk and results in an increase of approximately 30% in worst-case throughput. The disk is divided into three[5] zones R, S and T, and each zone into large, fixed-sized regions $(R_1,\ldots,R_n),(S_1,\ldots,S_n),(T_1,\ldots,T_n)$. These regions are then used in the following order: $(R_1,S_1,T_n,R_2,S_2,T_{n-1},\ldots,R_n,S_n,T_1)$; data is written sequentially to blocks within each region. The effective throughput is thus the average of throughput at 3 different points on the disk, and close to constant.

When accessing the disk sequentially, a zone-to-zone seek will be required after each region; the region size must thus be chosen to balance seek overhead with buffering requirements. For disks used in our experiments, a region size of 64MB results in one additional seek per second (degrading disk performance by less than 1%) at a buffering requirement of 16MB per device.

5 Indexing Archived Data

An Hyperion monitor needs to maintain an index which supports efficient retrospective queries, but also which may be created at high speed. Disk performance significantly limits the options available for the index; although minimizing random disk operations is a goal in any database, here multiple fields must be indexed in records arriving at a rate of over 100,000 per second per link. To scale to these rates, Hyperion relies on index structures that can be computed online and then *stored immutably*. Hyperion partitions a stream into intervals and computes one or more *signatures* [13] for each interval. The signatures can be tested for the presence of a record with a certain key in the associated data interval. Unlike a traditional B-tree-like structure, a signature only indicates whether a record matching a certain key is present; it does not indicate where in the interval that record is present. Thus, the entire interval needs to be retrieved and scanned for the result. However, if the key is not present, the entire interval can be skipped.

Signature indices are computed on a per-interval basis; no stream-wide index is maintained. This organization provides an index which may be streamed to disk along with the data—once all data within an interval have been examined (and streamed to storage), the signature itself can also be streamed out and a new signature computation begun for the next interval. This also solves the problem of removing keys from the index as they age out, as the signature associated with a data interval ages out as well.

[5] The original choice of 3 regions was selected experimentally, but later work [8] demonstrates that this organization results in throughput variations of less than 4% across inner-to-outer track ratios up to 4:1.

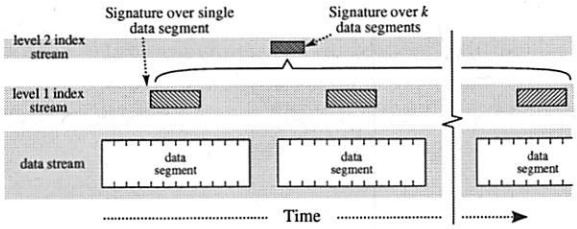


Figure 7: Hyperion multi-level signature index, showing two levels of signature index plus the associated data records.

5.1 Multi-level Signature Indices

Hyperion uses a multi-level *signature index*, the organization of which is shown in detail in Figure 7. A signature index, the most well-known of which is the Bloom Filter [3], creates a compact signature for one or more records, which may be tested to determine whether a particular key is present in the associated records. (This is in contrast to e.g. a B-tree or conventional hash table, where the structure provides a map from a key to the location where the corresponding record is stored.) To search for records containing a particular key, we first retrieve and test only the signatures; if any signature matches, then the corresponding records are retrieved and searched.

Signature functions are typically inexact, with some probability of a *false positive*, where the signature test indicates a match when there is none. This will be corrected when scanning the actual data records; the signature function cannot generate *false negatives*, however, as this will result in records being missed. Search efficiency for these structures is a trade-off between signature compactness, which reduces the amount of data retrieved when scanning the index, and false positive rate, which results in unnecessary data records being retrieved and then discarded.

The Hyperion index uses Bloom's hash function, where each key is hashed into a b-bit word, of which k bits are set to 1. The hash words for all keys in a set are logically OR-ed together, and the result is written as the signature for that set of records. To check for the presence of a particular key, the hash for that key h_0 is calculated and compared with the signature for the record, h_s; if any bit is set in h_0 but not set in h_s, then the value cannot be present in the corresponding data record. To calculate the false positive probability, we note that if the fraction of 1 bits in the signature for a set of records is r and the number of 1 bits in any individual hash is k, then the chance that a match could occur by chance is $1-(1-r)^k$; e.g. if the fraction of 1 bits is $\frac{1}{2}$, then the probability is 2^{-k}.

Multi-level index: Hyperion employs a two-level index [23], where a level-1 signature is computed for each data interval, and then a level-2 signature is computed over k data intervals. A search scans the level-2 signatures, and when a match is detected the corresponding k level-1 signatures are retrieved and tested; data blocks are retrieved and scanned only when a match is found in a level-1 signature.

When no match is found in the level-2 signature, k data segments may be skipped; this allows efficient search over large volumes of data. The level-2 signature will suffer from a higher false positive rate, as it is k times more concise than the level-1 signature; however, when a false positive occurs it is almost always detected after the retrieval of the level-1 signatures. In effect, the multi-level structure allows the compactness of the level-2 signature, with the accuracy of the level-1 signature.

Bit-sliced index: The description thus far assumes that signatures are streamed to disk as they are produced. When reading the index, however, a signature for an entire interval—thousands of bytes—must be retrieved from disk in order to examine perhaps a few dozen bits.

By buffering the top-level index and writing it in *bit-sliced* [12] fashion we are able to retrieve only those bits which need to be tested, thus possibly reducing the amount of data retrieved by orders of magnitude. This is done by aggregating N signatures, and then writing them out in N-bit *slices*, where the ith slice is constructed by concatenating bit i from each of the N signatures. If N is large enough, then a slice containing N bits, bit i from each of N signatures, may be retrieved in a single disk operation. (although not implemented at present, this is a planned extension to our system.)

5.2 Handling Range and Rank Queries

Although signature indices are very efficient, like other hash indices they are useful for exact-match queries only. In particular, they do not efficiently handle certain query types, such as range and rank (top-K) queries, which are useful in network monitoring applications.

Hyperion can use certain other functions as indices, as well. Two of these are interval bitmaps [26] and aggregation functions.

Interval bitmaps are a form of what are known as bitmap indices [4]; the domain of a variable is divided into b intervals, and a b-bit signature is generated by setting the one bit corresponding to the interval containing the variable's value. These signatures may then be superimposed, giving a summary which indicates whether a value within a particular range is present in the set of summarized records.

Aggregate functions such as *min* and *max* may be used as indexes as well; in this case the aggregate is calculated over a segment of data and stored as the signature for that data. Thus a query for $x < X_0$ can use aggregate minima to skip segments of data where no value will match, and a query for x with $COUNT(x) > N_0$ can make use of an

index indicating the top K values [17] in each segment and their counts.

Of these, *min* and *max* have been implemented in the Hyperion system.

5.3 Distributed Index and Query

Our discussion thus far has focused on data archiving and indexing locally on each node. A typical network monitoring system will comprise multiple nodes and it is necessary to handle distributed queries without resorting to query flooding. Hyperion maintains a distributed index that provides an integrated view of data at all nodes, while storing the data itself and most index information locally on the node where it was generated. Local storage is emphasized for performance reasons, since local storage bandwidth is more economical than communication bandwidth; storage of archived data which may never be accessed is thus most efficiently done locally.

To create this distributed index, a coarse-grain summary of the data archived at each node is needed. The top level of the Hyperion multi-level index provides such a summary, and is shared by each node with the rest of the system. Since broadcasting the index to all other nodes would result in excessive traffic as the system scales, an *index node* is designated for each time interval $[t_1, t_2)$. All nodes send their top-level indices to the index node during this time-interval. Designating a different index node for successive time intervals results in a temporally-distributed index. Cross-node queries are first sent to an index node, which uses the coarse-grain index to determine the nodes containing matching data; the query is then forwarded to this subset for further processing.

6 Implementation

We have implemented a prototype of the Hyperion network monitoring system on Linux, running on commodity servers; it currently comprises 7000 lines of code.

The StreamFS implementation takes advantage of Linux asynchronous I/O and raw device access, and is implemented as a user-space library. In an additional simplification, the file system root resides in a file on the conventional file system, rather than on the device itself. These implementation choices impose several constraints: for instance, all access to a StreamFS volume must occur from the same process, and that process must run as root in order to access the storage hardware. These limitations have not been an issue for Hyperion to date; however, a kernel implementation of StreamFS is planned which will address them.

The index is a two-level signature index with linear scan of the top level (not bit-sliced) as described in Section 5.1. Multiple keys may be selected to be indexed on, where each key may be a single field or a composite key consisting of multiple fields. Signatures for each key are then superimposed in the same index stream via logical OR. Query planning is not yet implemented, and the query API requires that each key to be used in performing the query be explicitly identified.

Packet input is supported from trace files and via a special-purpose gigabit ethernet driver, sk98_fast, developed for nProbe at the University of Cambridge [19]. Support for Endace DAG hardware is planned, as well.

The Hyperion system is implemented as a set of modules which may be controlled from a scripting language (Python) through an interface implemented via the SWIG wrapper toolkit. This design allows the structure of the monitoring application to be changed flexibly, even at run time—as an example, a query is processed by instantiating data source and index search objects and connecting them. Communication between Hyperion systems is by RPC, which allows remote query execution or index distribution to be handled and controlled by the same mechanisms as configuration within a single system.

7 Experimental Results

In this section we present operational measurements of the Hyperion network monitor system. Tests of the stream file system component, StreamFS, measure its performance and compare it to that of solutions based on general-purpose file systems. Micro-benchmarks as well as off-line tests on real data are used to test the multi-level indexing system; the micro-benchmarks measure the scalability of the algorithm, while the trace-based tests characterize the search performance of our index on real data. Finally, system experiments characterize the performance of single Hyperion nodes, as well as demonstrating operation of a multi-node configuration.

7.1 Experimental Setup

Unless specified otherwise, tests were performed on the following system:

Hardware	2 × 2.4GHz P4 Xeon, 1 GB memory
Storage	4 × Fujitsu MAP3367NP Ultra320 SCSI, 10K RPM
OS	Linux 2.6.9 (CentOS 4.4)
Network	SysKonnect SK-9821 1000mbps

File system tests wrote dummy data (i.e. zeros), and ignored data from read operations. Most index tests, however, used actual trace data from the link between the University of Massachusetts and the commercial Internet [31]. These trace files were replayed on another system by combining the recorded headers (possibly after modification) with dummy data, and transmitting the resulting packets directly to the system under test.

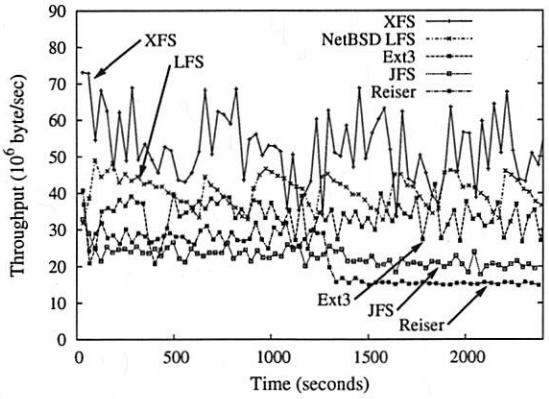


Figure 8: Streaming write-only throughput by file system. Each trace shows throughput for 30s intervals over the test run.

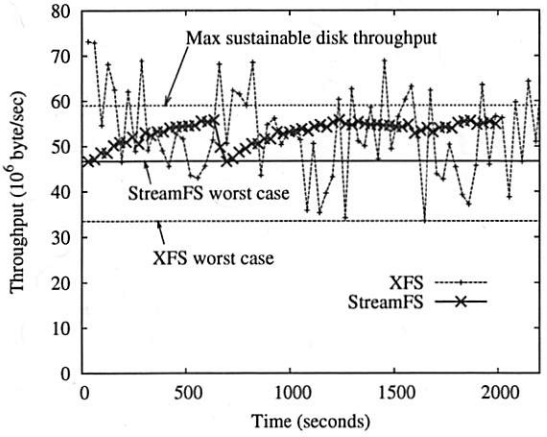


Figure 9: XFS vs. StreamFS write only throughput, showing 30 second and mean values. Straight lines indicate disk throughput at outer tracks (max) and inner tracks (min) for comparison. Note the substantial difference in worst-case throughput between the two file systems.

7.2 File Systems and Databases

Our first tests establish a baseline for evaluating the performance of the Hyperion system. Since Hyperion is an application-specific database, built on top of an application-specific file system, we compare its performance with that of existing general-purpose versions of these components. In particular, we measure the speed of storing network traces on both a conventional relational database and on several conventional file systems.

Database Performance: We briefly present results of bulk loading packet header data on Postgres 7.4.13. Approximately 14.5M trace data records representing 158 seconds of sampled traffic were loaded using the COPY command; after loading, a query retrieved a unique row in the table. To speed loading, no index was created, and no attempt was made to test simultaneous insert and query performance. Mean results with 95% confidence intervals (8 repetitions) are as follows:

data set	158 seconds	14.5M records
table load	252 s (±7 s)	1.56 × real-time
query	50.7 s (±0.4 s)	0.32 × real-time

Postgres was not able to load the data, collected on a moderately loaded (40%) link, in real time. Query performance was much too slow for on-line use; although indexing would improve this, it would further degrade load performance.

Baseline File system measurements: These tests measure the performance of general-purpose file systems to serve as the basis for an application-specific stream database for Hyperion. In particular, we measure write-only performance with multiple streams, as well as the ability to deliver write performance guarantees in the presence of mixed read and write traffic. The file systems tested on Linux are ext3, ReiserFS, SGI's XFS [29], and IBM's JFS; in addition LFS was tested on NetBSD 3.1.

Preliminary tests using the naïve single file per stream strategy from Section 4.1 are omitted, as performance for all file systems was poor. Further tests used an implementation of the log file strategy from Section 4.1, with file size capped at 64MB. Tests were performed with 32 parallel streams of differing speeds, with random write arrivals of mean size 64KB. All results shown are for the steady state, after the disk has filled and data is being deleted to make room for new writes.

The clear leader in performance is XFS, as may be seen in Figure 8. It appears that XFS maintains high write performance for a large number of streams by buffering writes and writing large extents to each file – contiguous extents as large as 100MB were observed, and (as expected) the buffer cache expanded to use almost all of memory during the tests. (Sweeney *et al.* [29] describe how XFS defers block assignment until pages are flushed, allowing such large extents to be generated.)

LFS has the next best performance. We hypothesize that a key factor in its somewhat lower performance was the significant overhead of the segment cleaner. Although we were not able to directly measure I/O rates due to cleaning, the system CPU usage of the cleaner process was significant: approximately 25% of that used by the test program.

7.3 StreamFS Evaluation

In light of the above results, we evaluate the Hyperion file system StreamFS by comparing it to XFS.

StreamFS Write Performance: In Figure 9 we see representative traces for 32-stream write-only traffic for StreamFS and XFS. Although mean throughput for both file systems closely approaches the disk limit, XFS shows high variability even when averaged across 30 second intervals. Much of the XFS performance variability remains within the range of the disk minimum and maximum throughput, and is likely due to allocation of large

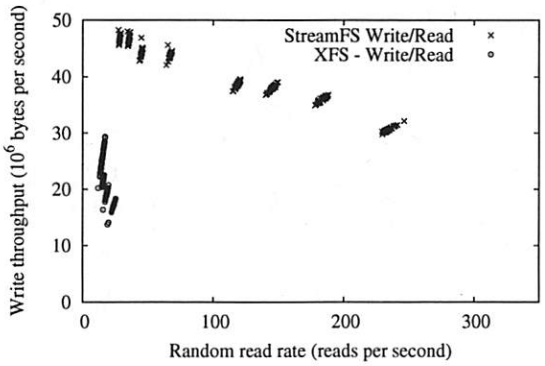


Figure 10: Scatter plot of StreamFS and XFS write and read performance.

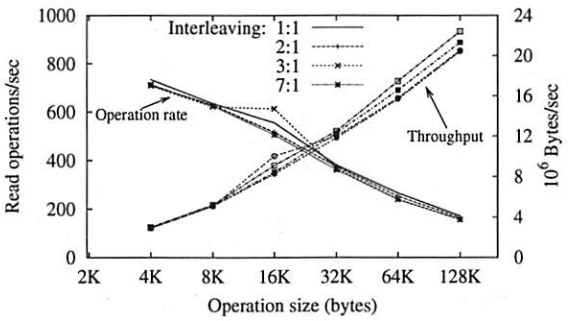


Figure 11: Streaming file system read performance. Throughput (rising) and operations/sec (falling) values are shown. The interleave factor refers to the number of streams interleaved on disk - i.e. for the 1:1 case the stream being fetched is the only one on disk; in the 1:7 case it is one of 7.

extents at random positions across the disk. A number of 30s intervals, however, as well as two 60s intervals, fall considerably below the minimum disk throughput; we have not yet determined a cause for these drop-offs in performance. The consistent performance of StreamFS, in turn, gives it a worst-case speed close to the mean — almost 50% higher than the worst-case speed for XFS.

Read/Write Performance: Useful measurements of combined read/write performance require a model of read access patterns generated by the Hyperion monitor. In operation, on-line queries read the top-level index, and then, based on that index, read non-contiguous segments of the corresponding second-level index and data stream. This results in a read access pattern which is highly non-contiguous, although most seeks are relatively small. We model this non-contiguous access stream as random read requests of 4KB blocks in our measurements, with a fixed ratio of read to write requests in each experiment.

Figure 10 shows a scatter plot of XFS and StreamFS performance for varying read/write ratios. XFS read performance is poor, and write performance degrades precipitously when read traffic is added. This may be a side effect of organizing data in logfiles, as due to the large

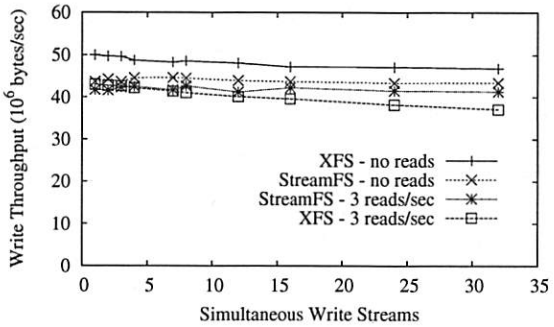


Figure 12: Sensitivity of performance to number of streams.

number of individual files, many read requests require opening a new file handle. It appears that these operations result in flushing some amount of pending work to disk; as evidence, the mean write extent length when reads are mixed with writes is a factor of 10 smaller than for the write-only case.

StreamFS Read Performance: We note that our prototype of StreamFS is not optimized for sequential read access; in particular, it does not include a read-ahead mechanism, causing some sequential operations to incur the latency of a full disk rotation. This may mask smaller-scale effects, which could come to dominate if the most significant overheads were to be removed.

With this caveat, we test single-stream read operation, to determine the effect of record size and stream interleaving on read performance. Each test writes one or more streams to an empty file system, so that the streams are interleaved on disk. We then retrieve the records of one of these streams in sequential order. Results may be seen in Figure 11, for record sizes ranging from 4KB to 128KB. Performance is dominated by per-record overhead, which we hypothesize is due to the lack of read-ahead mentioned above, and interleaved traffic has little effect on performance.

Sensitivity testing: These additional tests measured changes in performance with variations in the number of streams and of devices. Figure 12 shows the performance of StreamFS and XFS as the number of simultaneous streams varies from 1 to 32, for write-only and mixed read/write operations. XFS performance is seen to degrade slightly as the number of streams increases, and more so in the presence of read requests, while StreamFS throughput is relatively flat.

Multi-device tests with StreamFS on multiple devices and XFS over software RAID show almost identical speedup for both. Performance approximately doubles when going from 1 to 2 devices, and increases by a lesser amount with 3 devices as we get closer to the capacity of the 64-bit PCI bus on the test platform.

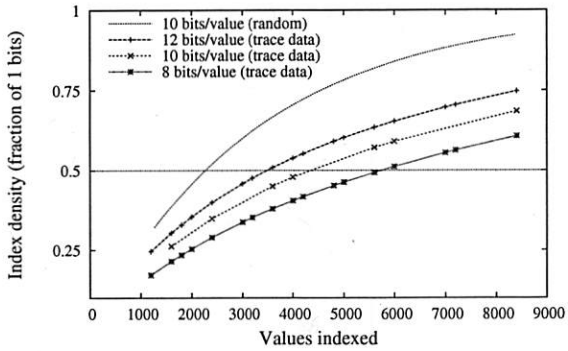


Figure 13: Signature density when indexing source/destination addresses, vs. number of addresses hashed into a single 4KB index block. Curves are for varying numbers of bits (k) per hashed value; top curve is for uniform random data, while others use sampled trace data.

7.4 Index Evaluation

The Hyperion index must satisfy two competing criteria: it must be fast to calculate and store, yet provide efficient query operation. We test both of these criteria, using both generated and trace data.

Signature Index Computation: The speed of this computation was measured by repeatedly indexing sampled packet headers in a large ($\gg$ cache size) buffer. on a single CPU. Since the size of the computation input — i.e. the number of headers indexed – is variable, linear regression was used to determine the relationship between computation parameters and performance.

In more detail, for each packet header we create N indices, where each index i is created on F_i fields (e.g. source address) totaling B_i bytes. For index i, the B_i bytes are hashed into an M-bit value with k bits set, as described in Section 5.1. Regression results for the significant parameters are:

variable	coeff.	std. error	t-stat
index (N)	132 ns	6.53 ns	20.2
bytes hashed (B_i)	9.4 ns	1.98 ns	4.72
bit generated (k)	43.5 ns	2.1 ns	21.1

As an example, if 7 indices are computed per header, with a total of 40 bytes hashed and 60 signature bits generated, then index computation would take $7 \cdot 132 + 40 \cdot 9.4 + 60 \cdot 43.5 = 3910$ns or 3.9 μs/packet, for a peak processing rate of 256,000 headers per second on the test CPU, a 2.4GHz XeonAlthough not sufficient to index minimum-sized packets on a loaded gigabit link, this is certainly fast enough for the traffic we have measured to date. (e.g. 106,000 packets/sec on a link carrying approximately 400Mbit/sec.)

Signature Density: This next set of results examines the performance of the Hyperion index after it has been written to disk, during queries. In Figure 13 we measure the signature *density*, or the fraction of bits set to 1, when summarizing addresses from trace data. On the X axis we see the number of addresses summarized in a single hash block, while the different traces indicate the

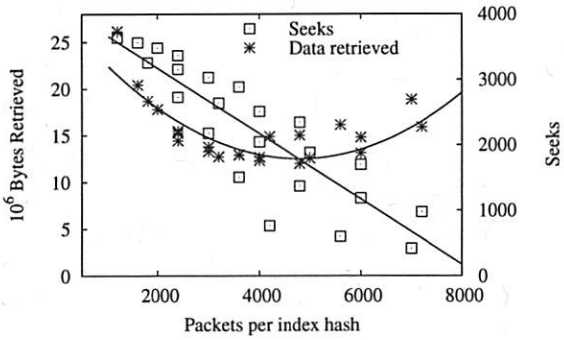


Figure 14: Single query overhead for summary index, with fitted lines. N packets (X axis) are summarized in a 4KB index, and then at more detail in several (3-6) 4KB sub-indices. Total read volume (index, sub-index, and data) and number of disk seeks are shown.

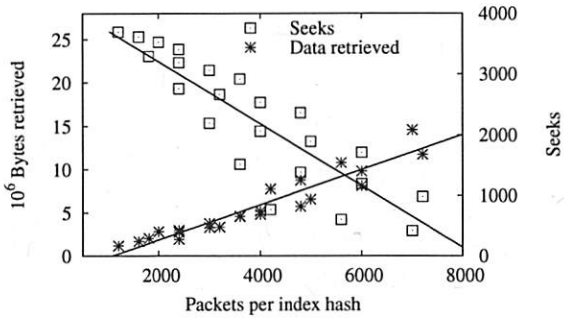


Figure 15: Single query overhead for bit-sliced index. Identical to Figure 14, except that each index was split into 1024 32-bit slices, with slices from 1024 indices stored consecutively by slice number.

precision with which each address is summarized. From Bloom [3] we know that the efficiency of this index is maximized when the fraction of 1 (or 0) bits is 0.5; this line is shown for reference.

From the graph we see that a 4KB signature can efficiently summarize between 3500 and 6000 addresses, depending on the parameter k and thus the false positive probability. The top line in the graph shows signature density when hashing uniformly-distributed random addresses with $k = 10$; it reaches 50% density after hashing only half as many addresses as the $k = 10$ line for real addresses. This is to be expected, due to repeated addresses in the real traces, and translates into higher index performance when operating on real, correlated data.

Query overhead: Since the index and data used by a query must be read from disk, we measure the overhead of a query by the factors which affect the speed of this operation: the volume of data retrieved and the number of disk seeks incurred. A 2-level index with 4K byte index blocks was tested, with data block size varying from 32KB to 96KB according to test parameters. The test indexed traces of 1 hour of traffic, comprising

26GB, $3.8 \cdot 10^8$ packets, and $2.5 \cdot 10^6$ unique addresses. To measure overhead of the index itself, rather than retrieval of result data, queries used were highly selective, returning only 1 or 2 packets.

Figures 14 and 15 show query overhead for the simple and bit-sliced indices, respectively. On the right of each graph, the volume of data retrieved is dominated by sub-index and data block retrieval due to false hits in the main index. To the left (visible only in Figure 14) is a domain where data retrieval is dominated by the main index itself. In each case, seek overhead decreases almost linearly, as it is dominated by skipping from block to block in the main index; the number of these blocks decreases as the packets per block increase. In each case there is a region which allows the 26GB of data to be scanned at the cost of 10-15 MB of data retrieved, and 1000-2000 disk seeks.

7.5 Prototype Evaluation

After presenting test results for the components of the Hyperion network monitor, we now turn to tests of its performance as a whole. Our implementation uses StreamFS as described above, and a 2-level index without bit-slicing. The following tests for performance, functionality, and scalability are presented below:

- performance tests: tests on single monitoring node which assess the system's ability to gather and index data at network speed, while simultaneously processing user queries.
- functionality testing: three monitoring nodes are used to trace the origin of simulated malicious traffic within real network data.
- scalability testing: a system of twenty monitoring nodes is used to gather and index trace data, to measure the overhead of the index update protocol.

Monitoring and Query Performance: These tests were performed on the primary test system, but with a single data disk. Traffic from our gateway link traces was replayed over a gigabit cable to the test system. First the database was loaded by monitoring an hour's worth of sampled data — $4 \cdot 10^8$ packets, or 26GB of packet header data. After this, packets were transmitted to the system under test with inter-arrival times from the original trace, but scaled so as to vary the mean arrival rate, with simultaneous queries. We compute packet loss by comparing the transmit count on the test system with the receive count on Hyperion, and measure CPU usage.

Figure 16 shows packet loss and free CPU time remaining as the packet arrival rate is varied.[6] Although

[6]Idle time is reported as the fraction of 2 CPUs which is available. Packet capture currently uses 100% of one CPU; future work should reduce this.

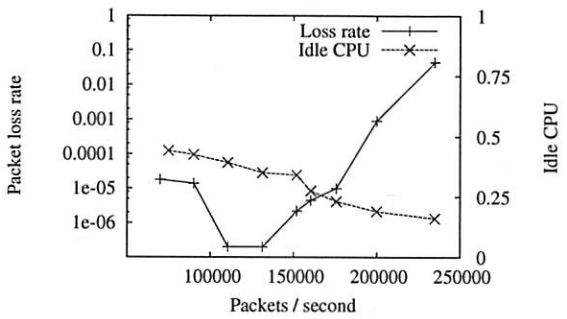


Figure 16: Packet arrival and loss rates.

loss rate is shown on a logarithmic scale, the lowest points represent zero packets lost out of 30 or 40 million received. The results show that Hyperion was able to receive and index over 200,000 packets per second with negligible packet loss. In addition, the primary resource limitation appears to be CPU power, indicating that it may be possible to achieve significantly higher performance as CPU speeds scale.

System Scalability: In this test a cluster of 20 monitors recorded trace information from files, rather than from the network itself. Tcpdump was used to monitor RPC traffic between the Hyperion processes on the nodes, and packet and byte counts were measured. Each of the 20 systems monitored a simulated link with traffic of approximately 110K pkts/sec, with a total bit rate per link of over 400 Mbit/sec. Level 2 indices were streamed to a *cluster head*, a position which rotates over time to share the load evenly. A third level of index was used in this test; each cluster head would store the indices received, and then aggregate them with its own level 2 index and forward the resulting stream to the current *network head*. Results are as follows:

Hyperion communication overhead overhead in K bytes/sec

	leaf	cluster head	net head
transmit	102 KB/s	102 KB/s	
receive		408 KB/s	510 KB/s

From these results we see that scaling to dozens of nodes would involve maximum traffic volumes between Hyperion nodes in the range of 4Mbit/s; this would not be excessive in many environments, such as within a campus.

Forensic Query Case Study: This experiment examines a simulated 2-stage network attack, based on real-world examples. Packet traces were generated for the attack, and then combined with sampled trace data to create traffic traces for the 3 monitors in this simulation, located at the campus gateway, the path to target A, and the path to B respectively.

Abbreviated versions of the queries for this search are as follows:

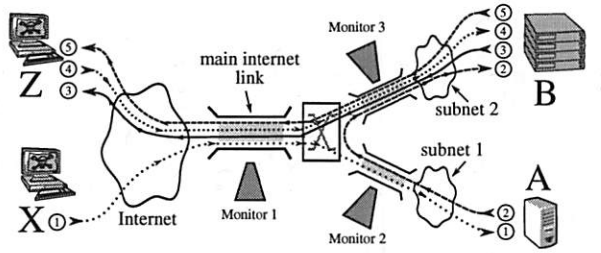


Figure 17: Forensic query example: (1) attacker X compromises system A, providing an inside platform to (2) attack system B, installing a bot. The bot (3) contacts system Z via IRC, (4) later receives a command, and begins (5) relaying spam traffic.

1 `SELECT p WHERE src=B,dport=SMTP,t≤`T_{now}
Search outbound spam traffic from B, locating start time T_0.

2 `SELECT p WHERE dst=B,t∈`$T_0 \cdots T_0 + \Delta$
Search traffic into B during single spam transmission to find control connection.

3 `SELECT p WHERE dst=B,t∈`$T_0 - \Delta \cdots T_0$
Find inbound traffic to B in the period before T_0.

4 `SELECT p WHERE s/d/p=Z/B/`P_x`, syn, t≤`T_0
Search for SYN packet on this connection at time T_{-1}.

5 `SELECT p WHERE dst=B,t∈`$T_{-1} - \Delta \cdots T_{-1}$
Search for the attack which infected B, finding connection from A at T_2.

6 `SELECT p WHERE dst=A,t∈`$T_{-2} - \Delta \cdots T_{-2} + \Delta$
Find external traffic to A during the A-B connection to locate attacker X.

7 `SELECT p WHERE src=X,syn,t≤`T_{-2}
Find all incoming connections from X

We note that additional steps beyond the Hyperion queries themselves are needed to trace the attack; for instance, in step 3 the search results are examined for patterns of known exploits, and the results from steps 5 and 6 must be joined in order to locate X. Performance of this search (in particular, steps 1, 4, and 7) depends on the duration of data to be searched, which depends in turn on how quickly the attack is discovered. In our test, Hyperion was able to use its index to handle queries over several hours of trace data in seconds. In actual usage it may be necessary to search several days or more of trace data; in this case the long-running queries would require minutes to complete, but would still be effective as a real-time forensic tool.

8 Related Work

Like Hyperion, both PIER [15] and MIND [18] are able to query past network monitoring data across a distributed network. Both of these systems, however, are based on DHT structures which are unable to sustain the high insertion rates required for indexing packet-level trace data, and can only index lower-speed sources such as flow-level information. The Gigascope [6] network monitoring system is able to process full-speed network monitoring streams, and provides a SQL-based query language. These queries, however, may only be applied over incoming data streams; there is no mechanism in GigaScope itself for *retrospective queries*, or queries over past data. StreamBase [28] is a general-purpose stream database, which like GigaScope is able to handle streaming queries at very high rates. In addition, like Hyperion, StreamBase includes support for persistent tables for retrospective queries, but these tables are conventional hash or B-tree-indexed tables, and are subject to the same performance limitations.

A number of systems such as the Endace DAG [10] have been developed for wire-speed collection and storage of packet monitoring data, but these systems are designed for off-line analysis of data, and provide no mechanisms for indexing or even querying the data. CoMo [16] addresses high-speed monitoring and storage, with provisions for both streaming and retrospective queries. Although it has a storage component, however, it does not include any mechanism for indexing, limiting its usefulness for querying large monitor traces.

The log structure of StreamFS bears a debt to the original Berkeley log-structured file system [22], as well as the WORM file system from the V System [5]. There has been much work in the past on supporting streaming reads and writes for multimedia file systems (e.g. [30]); however, the sequential-write random-read nature of our problem results in significantly different solutions.

There is an extensive body of literature on Bloom filters and the signature file index used in this system; two useful articles are a survey by Faloutsos [11] and the original article by Bloom [3]. Multi-level and multi-resolution indices have been described both in this body of literature (e.g. [23]) as well as in other areas such as sensor networks [14, 7].

9 Conclusions

In this paper, we argued that neither general-purpose file systems nor common database index structures meet the unique needs imposed by high-volume stream archiving and indexing. We proposed Hyperion, a novel stream archiving system that consists of StreamFS, a write-optimized stream file system, and a multi-level signature index that handles high update rates and enables distributed querying. Our prototype evaluation has shown that (i) StreamFS can scale to write loads of over a million packets per second, (ii) the index can support over 200K packet/s while providing good query performance for interactive use, and (iii) our system can scale to tens of monitors. As part of future work, we plan to enhance the aging policies in StreamFS and implement other index structures to support richer querying.

10 Acknowledgments

This work was funded in part by NFS grants EEC-0313747, CNS-0323597, and CNS-0325868. The authors would also like to thank Deepak Ganesan and the anonymous reviewers for contributions to this paper.

References

[1] ABADI, D. J., AHMAD, Y., BALAZINSKA, M., ÇETINTEMEL, U., CHERNIACK, M., HWANG, J.-H., LINDNER, W., MASKEY, A. S., RASIN, A., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONI, S. The Design of the Borealis Stream Processing Engine. In *Proc. Conf. on Innovative Data Systems Research* (Jan. 2005).

[2] BERNSTEIN, D. Syn cookies. Published at `http://cr.yp.to/syncookies.html`.

[3] BLOOM, B. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (July 1970), 422–426.

[4] CHAN, C.-Y., AND IOANNIDIS, Y. E. Bitmap index design and evaluation. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (June 1998), pp. 355–366.

[5] CHERITON, D. The V Distributed System. *Communications of the ACM 31*, 3 (Mar. 1988), 314–333.

[6] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: a stream database for network applications. In *Proc. 2003 ACM SIGMOD Intl. Conf. on Management of data* (2003), pp. 647–651.

[7] DESNOYERS, P., GANESAN, D., AND SHENOY, P. TSAR: a two tier sensor storage architecture using interval skip graphs. In *Proc. 3rd Intl. Conf on Embedded networked sensor systems (SenSys05)* (2005), pp. 39–50.

[8] DESNOYERS, P., AND SHENOY, P. Hyperion: High Volume Stream Archival for Retrospective Querying. Tech. Rep. TR46-06, University of Massachusetts, Sept. 2006.

[9] DREGER, H., FELDMANN, A., PAXSON, V., AND SOMMER, R. Operational experiences with high-volume network intrusion detection. In *Proc. 11th ACM Conf. on Computer and communications security (CCS '04)* (2004), pp. 2–11.

[10] ENDACE INC. Endace DAG4.3GE Network Monitoring Card. available at `http://www.endace.com`, 2006.

[11] FALOUTSOS, C. Signature-Based Text Retrieval Methods: A Survey. *IEEE Data Engineering Bulletin 13*, 1 (1990), 25–32.

[12] FALOUTSOS, C., AND CHAN, R. Fast Text Access Methods for Optical and Large Magnetic Disks: Designs and Performance Comparison. In *VLDB '88: Proc. 14th Intl. Conf. on Very Large Data Bases* (1988), pp. 280–293.

[13] FALOUTSOS, C., AND CHRISTODOULAKIS, S. Signature files: an access method for documents and its analytical performance evaluation. *ACM Trans. Inf. Syst. 2*, 4 (1984), 267–288.

[14] GANESAN, D., ESTRIN, D., AND HEIDEMANN, J. Dimensions: Distributed multi-resolution storage and mining of networked sensors. *ACM Computer Communication Review 33*, 1 (January 2003), 143–148.

[15] HUEBSCH, R., CHUN, B., HELLERSTEIN, J. M., LOO, B. T., MANIATIS, P., ROSCOE, T., SHENKER, S., STOICA, I., AND YUMEREFENDI, A. R. The Architecture of PIER: an Internet-Scale Query Processor. In *Proc. Conf. on Innovative Data Systems Research (CIDR)* (Jan. 2005).

[16] IANNACCONE, G., DIOT, C., MCAULEY, D., MOORE, A., PRATT, I., AND RIZZO, L. The CoMo White Paper. Tech. Rep. IRC-TR-04-17, Intel Research, Sept. 2004.

[17] KARP, R. M., SHENKER, S., AND PAPADIMITRIOU, C. H. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst. 28*, 1 (2003), 51–55.

[18] LI, X., BIAN, F., ZHANG, H., DIOT, C., GOVINDAN, R., HONG, W., AND IANNACCONE, G. Advanced Indexing Techniques for Wide-Area Network Monitoring. In *Proc. 1st IEEE Intl. Workshop on Networking Meets Databases (NetDB)* (2005).

[19] MOORE, A., HALL, J., KREIBICH, C., HARRIS, E., AND PRATT, I. Architecture of a Network Monitor. *Passive & Active Measurement Workshop 2003 (PAM2003)* (2003).

[20] MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., BABU, S., DATAR, M., MAKU, G., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. Query processing, approximation, and resource management in a data stream management system. In *Proc. Conf. on Innovative Data Systems Research (CIDR)* (2003).

[21] NDIAYE, B., NIE, X., PATHAK, U., AND SUSAIRAJ, M. A Quantitative Comparison between Raw Devices and File Systems for implementing Oracle Databases. `http://www.oracle.com/technology/deploy/performance/WhitePapers.html`, Apr. 2004.

[22] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[23] SACKS-DAVIS, R., AND RAMAMOHANARAO, K. A two level superimposed coding scheme for partial match retrieval. *Information Systems 8*, 4 (1983), 273–289.

[24] SELTZER, M. I., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. N. File System Logging versus Clustering: A Performance Comparison. In *USENIX Winter Technical Conference* (1995), pp. 249–264.

[25] SHRIVER, E., GABBER, E., HUANG, L., AND STEIN, C. A. Storage management for web proxies. In *Proc. USENIX Annual Technical Conference* (2001), pp. 203–216.

[26] STOCKINGER, K. Design and Implementation of Bitmap Indices for Scientific Data. In *Intl. Database Engineering & Applications Symposium* (July 2001).

[27] STONEBRAKER, M., CETINTEMEL, U., AND ZDONIK, S. The 8 requirements of real-time stream processing. *SIGMOD Record 34*, 4 (2005), 42–47.

[28] STREAMBASE, I. StreamBase: Real-Time, Low Latency Data Processing with a Stream Processing Engine. from `http://www.streambase.com`, 2006.

[29] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS File System. In *USENIX Annual Technical Conference* (Jan. 1996).

[30] TOBAGI, F. A., PANG, J., BAIRD, R., AND GANG, M. Streaming RAID: a disk array management system for video files. In *Proc. 1st ACM Intl. Conf. on Multimedia* (1993), pp. 393–400.

[31] Umass trace repository. Available at `http://traces.cs.umass.edu`.

[32] WANG, W., ZHAO, Y., AND BUNT, R. HyLog: A High Performance Approach to Managing Disk Layout. In *Proc. 3rd USENIX Conf. on File and Storage Technologies (FAST)* (2004), pp. 145–158.

Load Shedding in Network Monitoring Applications

Pere Barlet-Ros*, Gianluca Iannaccone†, Josep Sanjuàs-Cuxart*,
Diego Amores-López*, Josep Solé-Pareta*

* *Technical University of Catalonia (UPC)*
Barcelona, Spain

† *Intel Research*
Berkeley, CA

Abstract

Monitoring and mining real-time network data streams is crucial for managing and operating data networks. The information that network operators desire to extract from the network traffic is of different size, granularity and accuracy depending on the measurement task (e.g., relevant data for capacity planning and intrusion detection are very different). To satisfy these different demands, a new class of monitoring systems is emerging to handle multiple arbitrary and continuous traffic queries. Such systems must cope with the effects of overload situations due to the large volumes, high data rates and bursty nature of the network traffic.

In this paper, we present the design and evaluation of a system that can shed excess load in the presence of extreme traffic conditions, while maintaining the accuracy of the traffic queries within acceptable levels. The main novelty of our approach is that it is able to operate without explicit knowledge of the traffic queries. Instead, it extracts a set of features from the traffic streams to build an on-line prediction model of the query resource requirements. This way the monitoring system preserves a high degree of flexibility, increasing the range of applications and network scenarios where it can be used.

We implemented our scheme in an existing network monitoring system and deployed it in a research ISP network. Our results show that the system predicts the resources required to run each traffic query with errors below 5%, and that it can efficiently handle extreme load situations, preventing uncontrolled packet losses, with minimum impact on the accuracy of the queries' results.

1 Introduction

Network monitoring applications that must extract a large number of real-time metrics from many input streams are becoming increasingly common. These include for example applications that correlate network data from multiple sources (e.g., end-systems, access points, switches) to identify anomalous behaviors, enable traffic engineering and capacity planning or manage and troubleshoot the network infrastructure.

The main challenge in these systems is to keep up with ever increasing input data rates and processing requirements. Data rates are driven by the increase in network link speeds, application demands and the number of end-hosts in the network. The processing requirements are growing to satisfy the demands for fine grained and continuous analysis, tracking and inspection of network traffic. This challenge is made even harder as network operators expect the queries to return accurate enough results in the presence of extreme or anomalous traffic patterns, when the system is under additional stress (and the query results are most valuable!). The alternative of over-provisioning the system to handle peak rates or any possible traffic mix would be prohibitively expensive and result in a highly underutilized system based on an extremely pessimistic estimation of workload.

Recently, several research proposals have addressed this challenge [18, 22, 23, 8, 14]. The solutions introduced belong to two broad categories. The first includes approaches that consider a pre-defined set of metrics and can report approximate results (within given accuracy bounds) in the case of overload [18, 14]. The second category includes solutions that define a declarative query language with a small set of operators for which the resource usage is assumed to be known [22, 23, 8]. In the presence of overload, operator-specific load shedding techniques are implemented (e.g., selectively discarding some records, computing approximate summaries) so that the accuracy of the entire query is preserved within certain bounds. These solutions present two common limitations: (i) they restrict the types of metrics that can be extracted from the traffic streams, limiting therefore the possible uses and applications of these systems, and (ii) they assume explicit knowledge of the cost and selectivity of each operator, requiring a very careful and

time-consuming design and implementation phase for each of them.

In this paper, we present a system that supports multiple arbitrary and continuous traffic queries on the input streams. The system can handle overload situations due to anomalous or extreme traffic mixes by gracefully degrading the accuracy of the queries. The core of our load shedding scheme consists of the real-time modeling and prediction of the system resource usage that allows the system to *anticipate* future bursts in the resource requirements. The main novelty of our approach is that it does not require explicit knowledge of the query or of the types of computations it performs (e.g., flow classification, maintaining aggregate counters, string search). This way we preserve the flexibility of the monitoring system, enabling fast implementation and deployment of new network monitoring applications.

Without any knowledge of the computations performed on the packet streams, we infer their cost from the relation between a large set of pre-defined "features" of the input stream and the actual resource usage. A feature is a counter that describes a specific property of a sequence of packets (e.g., number of unique source IP addresses). The features we compute on the input stream have the advantage of being lightweight with a deterministic worst case computational cost. Then, we automatically identify those features that best model the resource usage of each query and use them to predict the overall load of the system. This short-term prediction is used to guide the system on deciding *when*, *where* and *how much* load to shed. In the presence of overload, the system can apply several load shedding techniques, such as packet sampling, flow sampling or computing summaries of the data streams to reduce the amount of resources required by the queries to run.

For simplicity, in this paper we focus only on one resource: the CPU cycles. Other system resources are also critical (e.g., memory, disk bandwidth and disk space) and we believe that approaches similar to what we propose here could be applied as well.

We have integrated our load shedding scheme into the CoMo monitoring system [16] and deployed it on a research ISP network, where the traffic load and query requirements exceed by far the system capacity. We ran a set of seven concurrent queries that range from maintaining simple counters (e.g., number of packets, application breakdown) to more complex data structures (e.g., per-flow classification, ranking of most popular destinations or pattern search).

Our results show that, with the load shedding mechanism in place, the system effectively handles extreme load situations, while being always responsive and preventing uncontrolled packet losses. The results also indicate that a predictive approach can quickly adapt to overload situations and keep the queries' results within acceptable error bounds, as compared to a reactive load shedding strategy.

The remainder of this paper is structured as follows. Section 2 presents in greater detail some related work. Section 3 introduces the monitoring system and the set of queries we use for our study. We describe our prediction method in Section 4 and validate its performance using real-world packet traces in Section 5. Section 6 presents a load shedding scheme based on our prediction method. Finally, in Section 7 we evaluate our load shedding scheme in a research ISP network, while Section 8 concludes the paper and introduces ideas for future work.

2 Related Work

The design of mechanisms to handle overload situations is a classical problem in any real-time system design and several previous works have proposed solutions in different environments.

In the network monitoring space, NetFlow [9] is considered the state-of-the-art. In order to handle the large volumes of data exported and to reduce the load on the router it resorts to packet sampling. The sampling rate must be defined at configuration time and network operators tend to set it to a low "safe" value (e.g., 1/100 or 1/1000 packets) to handle unexpected traffic scenarios. Adaptive NetFlow [14] allows routers to dynamically tune the sampling rate to the memory consumption in order to maximize the accuracy given a specific incoming traffic mix. Keys et al. [18] extend the approach used in NetFlow by extracting and exporting a set of 12 traffic summaries that allow the system to answer a fixed number of common questions asked by network operators. They deal with extreme traffic conditions using adaptive sampling and memory-efficient counting algorithms. Our work differs from these approaches in that we are not limited to a fixed set of known traffic reports, but instead we can handle arbitrary network traffic queries, increasing the range of applications and network scenarios where the monitoring system can be used.

Several research proposals in the stream database literature are also very relevant to our work. The Aurora system [5] can process a large number of concurrent queries that are built out of a small set of operators. In Aurora, load shedding is achieved by inserting additional drop operators in the data flow of each query [23]. In order to find the proper location to insert the drop operators, [23] assumes explicit knowledge of the cost and selectivity of each operator in the data flow. In [7, 22], the authors propose a system that applies approximate query processing techniques, instead of dropping records, to provide approximate and delay-bounded answers in presence of overload. On the contrary, in our context we have no ex-

plicit knowledge of the query and therefore we cannot make any assumption on its cost or selectivity to know when it is the right time to drop records. Regarding the records to be dropped, we apply packet or flow sampling to reduce the load on the system, but other summarization techniques are an important piece of future work.

In the Internet services space, SEDA [24] proposes an architecture to develop highly concurrent server applications, built as networks of stages interconnected by queues. SEDA implements a reactive load shedding scheme by dropping incoming requests when an overload situation is detected (e.g., the response time of the system exceeds a given threshold). In this work we use instead a predictive approach to anticipate overload situations. We will show later how a predictive approach can significantly reduce the impact of overload as compared to a reactive one.

Finally, our system is based on extracting features from the traffic streams with deterministic worst case time bounds. Several solutions have been proposed in the literature to this end. For example, counting the number of distinct items in a stream has been addressed in the past in [15, 1]. In this work we implement the multi-resolution bitmap algorithms for counting flows proposed in [15].

3 System Overview

The basic thesis behind this work is that the cost of maintaining the data structures needed to execute a query can be modeled by looking at a set of traffic features that characterizes the input data. The intuition behind this thesis comes from the empirical observation that each query incurs a different overhead when performing basic operations on the state it maintains while processing the input packet stream (e.g., creating new entries, updating existing ones or looking for a valid match). We observed that the time spent by a query is mostly dominated by the overhead of some of these operations and therefore can be modeled by considering the right set of simple traffic features.

A traffic feature is a counter that describes a property of a sequence of packets. For example, potential features could be the number of packets or bytes in the sequence, the number of unique source IP addresses, etc. In this paper we will select a large set of simple features that have the same underlying property: deterministic worst case computational complexity.

Once a large number of features is efficiently extracted from the traffic stream, the challenge is in identifying the right ones that can be used to accurately model and predict the query's CPU usage. Figure 1 illustrates a very simple example. The figure shows the time series of the CPU cycles consumed by an "unknown" query (top graph) when running over a $100s$ snapshot of our dataset (described in Section 5.1), where we inserted an artificially generated anomaly. The three bottom plots show three possible features over time: the number of packets, bytes and flows (defined by the classical 5-tuple: source and destination addresses, source and destination port numbers and protocol number). It is clear from the figure that the bottom plot would give us more useful information to predict the CPU usage over time for this query. It is also easy to infer that the query is performing some sort of per-flow classification, hence the higher cost when the number of flows increases, despite the volume of packets and bytes remains fairly stable.

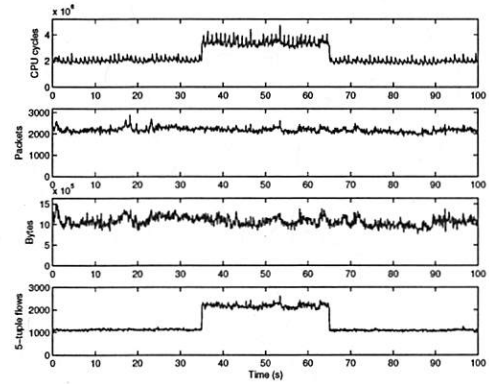


Figure 1: CPU usage of an "unknown" query in the presence of an artificially generated anomaly compared to the number of packets, bytes and flows

We designed a method that automatically selects the most relevant feature(s) from small sequences of packets and uses them to accurately predict the CPU usage of arbitrary queries. This fine-grained and short-term prediction is then used to quickly adapt to overload situations by sampling the input streams.

3.1 Monitoring Platform

We chose the CoMo platform [16] to develop and evaluate our resource usage prediction and load shedding methods. CoMo is an open-source passive monitoring system that allows for fast implementation and deployment of network monitoring applications. CoMo follows a modular approach where users can easily define traffic queries as plug-in modules written in C, making use of a feature-rich API provided by the core platform. Users are also required to specify a simple stateless filter to be applied on the incoming packet stream (it could be all the packets) as well as the granularity of the measurements, hereafter called *measurement interval* (i.e., the time interval that will be used to report continuous query results). All complex stateful computations are contained

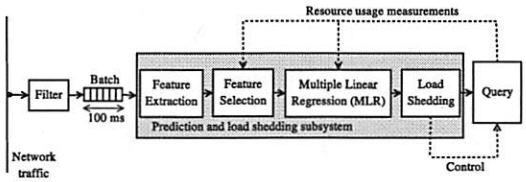


Figure 2: Prediction and load shedding subsystem

Name	Description
application	Port-based application classification
flows	Per-flow counters
high-watermark	High watermark of link utilization
link-count	Traffic load
pattern search	Identifies sequences of bytes in the payload
top destinations	Per-flow counters for the top-10 destination IPs
trace	Full-payload collection

Table 1: Queries used in the experimental evaluation

within the plug-in module code. This approach allows users to define traffic queries that otherwise could not be easily expressed using common declarative languages (e.g., SQL). More details about the CoMo platform can be found in [16].

In order to provide the user with maximum flexibility when writing queries, CoMo does not restrict the type of computations that a plug-in module can perform. As a consequence, the platform does not have any explicit knowledge of the data structures used by the plug-in modules or the cost of maintaining them. Therefore, any load shedding mechanism for such a system must operate only with external observations of the CPU requirements of the modules – and these are not known in advance but only after a packet has been processed.

Figure 2 shows the components and the data flow in the system. The prediction and load shedding subsystem (in gray) intercepts the packets from the filter before they are sent to the plug-in module implementing the traffic query. The system operates in four phases. First, it groups each $100ms$ of traffic in a "batch" of packets[1]. Each batch is then processed to extract a large predefined set of traffic features (Section 4.1). The feature selection subsystem is in charge of selecting the most relevant features according to the recent history of the query's CPU usage (Section 4.3). This phase is important to reduce the cost of the prediction algorithm, because it allows the system to discard beforehand the features regarded as useless for prediction purposes. This subset of relevant features is then given as input to the multiple linear regression subsystem to predict the CPU cycles required by the query to process the entire batch (Section 4.2). If the prediction exceeds the system capacity, the load shedding subsystem pre-processes the batch to discard (via packet or flow sampling) a portion of the packets (Section 6). Finally, the actual CPU usage is computed and fed back to the prediction subsystem to close the loop (Section 4.4).

3.2 Queries

Despite the fact that the actual metric computed by the query is not relevant for our work – our system considers all queries as black boxes – we are interested in considering a wide range of queries when performing the evaluation. We have selected the set of queries that are part of the standard distribution of CoMo[2]. Table 1 provides a brief summary of the queries. We believe that these queries form a representative set of typical uses of a real-time network monitoring system. They present different CPU usage profiles for the same input traffic and use different data structures to maintain their state (e.g., aggregated counters, arrays, hash tables, linked lists).

4 Prediction Methodology

In this section we describe in detail the three phases that our system executes to perform the prediction (i.e., *feature extraction*, *feature selection* and *multiple linear regression*) and how the resource usage is monitored. The only information we require from the continuous query is the measurement interval of the results. Avoiding the use of additional information increases the range of applications where this approach can be used and also reduces the likelihood of compromising the system by providing incorrect information about a query.

4.1 Feature Extraction

We are interested in finding a set of traffic features that are simple and inexpensive to compute, while helpful to characterize the CPU usage of a wide range of queries. A feature that is too specific may allow us to predict a given query with great accuracy, but could have a cost comparable to directly answering the query (e.g., counting the packets that contain a given pattern to predict the cost of signature-based IDS-like queries). Our goal is therefore to find features that may not explain in detail the entire cost of a query, but can provide enough information about the aspects that dominate the processing cost. For instance, in the previous example of a signature-based IDS query, the cost of matching a string will mainly depend on the number of collected bytes.

In addition to the number of packets and bytes, we maintain four counters per *traffic aggregate* that are updated every time a batch is received. A traffic aggregate considers one or more of the TCP/IP header fields: source and destination IP addresses, source and destination port numbers and protocol number. The four coun-

No.	Traffic aggregate
1	src-ip
2	dst-ip
3	protocol
4	<src-ip, dst-ip>
5	<src-port, proto>
6	<dst-port, proto>
7	<src-ip, src-port, proto>
8	<dst-ip, dst-port, proto>
9	<src-port, dst-port, proto>
10	<src-ip, dst-ip, src-port, dst-port, proto>

Table 2: Set of traffic aggregates (built from combinations of TCP/IP header fields) used by the prediction

ters we monitor per aggregate are: (i) the number of unique items in the batch; (ii) the number of new items compared to all items seen in a measurement interval; (iii) the number of repeated items in the batch (i.e., items in the batch minus unique) and (iv) the number of repeated items compared to all items in a measurement interval (i.e., items in the batch minus new).

For example, we may aggregate packets based on the source IP address and source port number, and then count the number of unique, new and repeated source IP address and source port pairs. Table 2 shows the combinations of the five header fields considered in this work. Although we do not evaluate other choices here, we note that other aggregates may also be useful (e.g., source IP prefixes or other combinations of the 5 header fields). Adding new traffic features (e.g., payload-related features) as well as considering other combinations of the existing ones is an important part of our future work.

This large set of features (four counters per traffic aggregate plus the total packet and byte counts, i.e., 42 in our experiments) helps narrow down which basic operations performed by the queries dominate their processing costs (e.g., creating a new entry, updating an existing one or looking up entries). For example, the new items are relevant to predict the CPU requirements of those queries that spend most time creating entries in the data structures, while the repeated items feature may be relevant to queries where the cost of updating the data structures is much higher than the cost of creating entries.

In order to extract the features with minimum overhead, we implement the multi-resolution bitmap algorithms proposed in [15]. The advantage of the multi-resolution bitmaps is that they bound the number of memory accesses per packet as compared to classical hash tables and they can handle a large number of items with good accuracy and smaller memory footprint than linear counting [25] or bloom filters [4]. We dimension the multi-resolution bitmaps to obtain counting errors around 1% given the link speeds in our testbed.

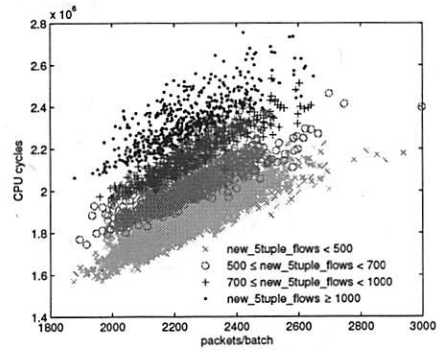


Figure 3: Scatter plot of the CPU usage versus the number of packets in the batch (*flows* query)

4.2 Multiple Linear Regression

Regression analysis is a widely applied technique to study the relationship between a response variable Y and one or more predictor variables $X_1, X_2, \ldots, X_p$. The linear regression model assumes that the response variable Y is a linear function of the p X_i predictor variables[3]. The fact that this relationship exists can be exploited for predicting the expected value of Y (i.e., the CPU usage) when the values of the p predictor variables (i.e., the individual features) are known.

When only one predictor variable is used, the regression model is often referred to as simple linear regression (SLR). Using just one predictor has two major drawbacks. First, there is no single predictor that yields good performance for all queries. For example, the CPU usage of the *link-count* query can be well modeled by looking at the number of packets in the batch, while the *trace* query would be better modeled by the number of bytes. Second, the CPU usage of more complex queries may depend on more than a single feature. To illustrate this latter point, we plot in Figure 3 the CPU usage for the *flows* query versus the number of packets in the batch. As we can observe, there are several underlying trends that depend both on the number of packets and on the number of new 5-tuples in the batch. This behavior is due to the particular implementation of the *flows* query that maintains a hash table to keep track of the flows and expires them at the end of each measurement interval.

Multiple linear regression (MLR) extends the simple linear regression model to several predictor variables. The general form of a linear regression model for p predictor variables can be written as follows [10]:

$$Y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \ldots \\ \cdots + \beta_p X_{pi} + \varepsilon_i, \qquad i = 1, 2, \ldots, n. \tag{1}$$

In fact, Equation 1 corresponds to a system of equations

that in matrix notation can be written as:

$$Y = X\beta + \varepsilon \quad (2)$$

where Y is a $n \times 1$ column vector of the response variable observations (i.e., the CPU usage of the previous n batches processed by the query); X is a $n \times (p+1)$ matrix resulting from n observations of the p predictor variables $X_1, \ldots, X_p$ (i.e., the values of the p features extracted from the previous n batches) with a first column of 1's that represents the intercept term β_0; β is a $(p+1) \times 1$ column vector of unknown parameters $\beta_0, \beta_1, \ldots, \beta_p$ ($\beta_1, \ldots, \beta_p$ are referred to as the *regression coefficients* or *weights*); and ε is a $n \times 1$ column vector of n residuals ε_i.

The estimators b of the regression coefficients β are obtained by the Ordinary Least Squares (OLS) procedure, which consists of choosing the values of the unknown parameters $b_0, \ldots, b_p$ in such a way that the sum of squares of the residuals is minimized. In our implementation, we use the singular value decomposition (SVD) method [21] to compute the OLS. Although SVD is more expensive than other methods, it is able to obtain the best approximation, in the least-squares sense, in the case of an over- or underdetermined system.

The statistical properties of the OLS estimators lie on some assumptions that must be fulfilled [10, pp. 216]: (i) the rank of X is $p+1$ and is less than n, i.e., there are no exact linear relationships among the X variables (no *multicollinearity*); (ii) the variable ε_i is normally distributed and the expected value of the vector ε is zero; (iii) there is no correlation between the residuals and they exhibit constant variance; (iv) the covariance between the predictors and the residuals is zero. In Section 4.3 we present a technique that makes sure the first assumption is valid. We have also verified experimentally using the packet traces of our dataset that the other assumptions hold but in the interest of space we will not show the results here.

4.3 Feature Selection

Since we assume arbitrary queries, we cannot know in advance which features should be used as predictors in the MLR for each query. Including all the extracted traffic features in the regression has several drawbacks: (i) the cost of the linear regression increases quadratically with the number of predictors, much faster than the gain in terms of accuracy (*irrelevant predictors*); (ii) even including all possible predictors, there would still be a certain amount of randomness that cannot be explained by any predictor; (iii) predictors that are linear functions of other predictors (*redundant predictors*) invalidate the no multicollinearity assumption[4].

It is therefore important to identify a small subset of features to be used as predictors. In order to support arbitrary queries, we need to define a generic feature selection algorithm. We would also like our method to be capable of dynamically selecting different sets of features if the traffic conditions change during the execution, and the current prediction model becomes obsolete.

Most of the algorithms proposed in the literature are based on a sequential variable selection procedure [10]. However, they are usually too expensive to be used in a real-time system. For this reason, we decided to use a variant of the Fast Correlation-Based Filter (FCBF) [26], which can effectively remove both irrelevant and redundant features and is computationally very efficient. Our variant differs from the original FCBF algorithm in that we use the *linear correlation coefficient* as a predictor goodness measure, instead of the *symmetrical uncertainty* measure used in [26].

The algorithm consists of two main phases. First, the linear correlation coefficient between each predictor and the response variable is computed and the predictors with a coefficient below a pre-defined *FCBF threshold* are discarded as not relevant. In Section 5.2 we will address the problem of choosing the appropriate FCBF threshold. Second, the predictors that are left after the first phase are ranked according to their coefficient values and processed iteratively to discard redundant predictors (i.e., predictors that have a mutual strong correlation), as described in [3]. The overall complexity of the FCBF is $O(n\,p \log p)$, where n is the number of observations and p the number of predictors [26].

4.4 Measurement of System Resources

Fine grained measurement of CPU usage is not an easy task. The mechanisms provided by the operating system do not offer enough resolution for our purposes, while processor performance profiling tools [17] impose a large overhead and are not a viable permanent solution.

In this work, we use instead the *time-stamp counter* (TSC) to measure the CPU usage, which is a 64-bit counter incremented by the processor every clock cycle [17]. In particular, we read the TSC before and after a batch is processed by a query. The difference between these two values corresponds to the number of CPU cycles used by the query to process the batch.

The CPU usage measurements that are fed back to the prediction system should be accurate and free of external noise to reduce the errors in the prediction. However, we empirically detected that measuring CPU usage at very small timescales incurs in several sources of noise:

Instruction reordering. The processor can reorder instructions at run time in order to improve performance.

In practice, the `rdtsc` instruction used to read the TSC counter is often reordered, since it simply consists of reading a register and it has no dependencies with other instructions. To avoid the effects of reordering, we execute a serializing instruction (e.g., `cpuid`) before and after our measurements [17]. Since the use of serializing instructions can have a severe impact on the system performance, we only take two TSC readings per query and batch, and we do not take any partial measurements during the execution of the query.

Context switches. The operating system may decide to schedule out the query process between two consecutive readings of the TSC. In that case, we would be measuring not only cycles belonging to the query, but also cycles of the process (or processes) that are preempting the query.

In order to avoid degrading the accuracy of future predictions when a context switch happens during a measurement, we discard those observations from the history and replace them with our prediction. To measure context switches, we monitor the *rusage* process structure in the Linux kernel.

Disk accesses. Disk accesses can interfere with the CPU cycles needed to process a query. In CoMo, a separate process is responsible for scheduling disk accesses to read and write query results. In practice, since disk transfers are done asynchronously by DMA, memory accesses of queries have to compete for the system bus with disk transfers. For the interested reader we show the limited impact of disk accesses on the prediction accuracy in [3].

It is important to note that all the sources of noise we detected so far are independent from the input traffic. Therefore, they cannot be exploited by a malicious user trying to introduce errors in our CPU measurements to attack the monitoring system.

5 Validation

In this section we show the performance of our prediction method on real-world traffic traces. In order to understand the impact of each parameter, we study the prediction subsystem in isolation from the sources of measurement noise identified in Section 4.4. We disabled the disk accesses in the CoMo process responsible for storage operations to avoid competition for the system bus. In Section 7, we will evaluate our method in a fully operational system.

To measure the performance of our method we consider the relative error in the CPU usage prediction while executing the seven queries defined in Table 1 over the traces in our dataset. The relative error is defined as the absolute value of one minus the ratio of the prediction and the actual number of CPU cycles spent by the queries over each batch. A more detailed performance analysis can be found in [3].

Trace name	Date	Time	Pkts (M)	Link load (Mbps) mean/max/min
w/o payloads	02/Nov/05	4:30pm-5pm	103.7	360.5/483.3/197.3
with payloads	11/Apr/06	8am-8:30am	49.4	133.0/212.2/096.1

Table 3: Traces used in the validation

5.1 Dataset

We collected two 30-minute traces from one direction of the Gigabit Ethernet link that connects the Catalan Research and Education Network (Scientific Ring) to the global Internet via its Spanish counterpart (RedIRIS). The Scientific Ring is managed by the Supercomputing Center of Catalonia (CESCA) and connects more than fifty Catalan universities and research centers using many different technologies that range from ADSL to Gigabit Ethernet [19]. A trace collected at this capture point is publicly available in the NLANR repository [20].

The first trace contains only packet headers, while the second one includes the entire packet payloads instead. Details of the traces are presented in Table 3.

5.2 Prediction Parameters

In our system, two configuration parameters impact the cost and accuracy of the predictions: the number of observations (i.e., n or the "history" of the system) and the FCBF threshold used to select the relevant features.

Number of observations. Figure 4 shows the average cost of computing the MLR versus the prediction accuracy over multiple executions, with values of history ranging from $1s$ to $100s$ (i.e., 10 to 1000 batches). As we can see, the cost grows linearly with the amount of history, since every additional observation translates into a new equation in the system in (2). The relative error between the prediction and the actual number of CPU cycles spent by the queries stabilizes around 1.2% for histories longer than 6 seconds. Larger errors for very small amounts of history (e.g., $1s$) are due to the fact that the number of predictors (i.e., $p = 42$) is larger than the amount of history (i.e., $n = 10$ batches) and thus the no multicollinearity assumption is not met. We also checked that histories longer than $100s$ do not improve the accuracy, because events that are not modeled by the traffic features are probably contributing to the error. Moreover, a longer history makes the prediction model less responsive to sudden changes in the traffic that may change the behavior of a query. In the rest of the paper we use a number of observations equal to 60 batches (i.e., $6s$).

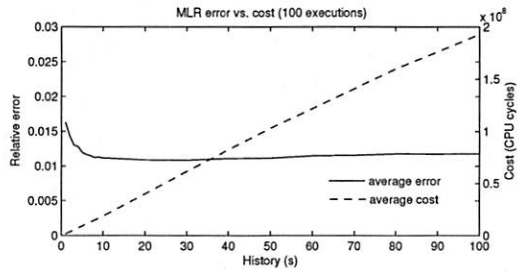


Figure 4: Prediction error versus cost as a function of the amount of history used to compute the MLR

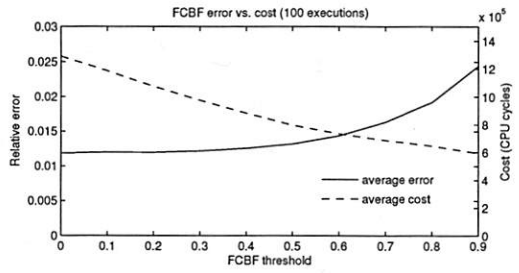


Figure 5: Prediction error versus cost as a function of the Fast Correlation-Based Filter threshold

FCBF threshold. The FCBF threshold determines which traffic features are relevant *and* not redundant in modeling the response variable. Figure 5 presents the prediction cost and accuracy as functions of the FCBF threshold over multiple executions in our testbed, with threshold values ranging from 0 (i.e., all features are considered relevant but the redundant ones are not selected) to 0.9 (i.e., most features are not selected). The prediction cost includes both the cost of the selection algorithm and the cost of computing the MLR with the selected features. Comparing this graph with Figure 4, we can see that using FCBF reduces the overall cost of the prediction by more than an order of magnitude while maintaining similar accuracy.

As the threshold increases, less predictors are selected, and this turns into a decrease in the CPU cycles needed to run the MLR. However, the error remains fairly close to the minimum value obtained when all features are selected, and starts to ramp up only for relatively large values of the threshold (around 0.6). Very large values of the threshold (above 0.8) experience a much faster increase in the error compared to the decrease in the cost. In the rest of the paper we use a value of 0.6 for the FCBF threshold that achieves a good trade-off between prediction cost and accuracy.

5.3 Prediction Accuracy

In order to evaluate the performance of our method we ran the seven queries of Table 1 over the two traces in our dataset. Figures 6 and 7 show the time series of the average and maximum error over five executions when running on the packet trace with and without payloads, respectively.

The average error in both cases is consistently below 2%, while the maximum error peaks around 10%. These larger errors are due to external system events unrelated to the traffic that cause a spike in the CPU usage (e.g., cache misses) or due to a sudden change in the traffic patterns that is not appropriately modeled by the features that the prediction is using at that time. However, the time series shows that our method is able to converge very quickly. The trace without payloads (Figure 7) exhibits better performance, with average errors that drop well below 1%.

In Table 4, we show the breakdown of the prediction errors by query. The average error is very low for each query, with a relatively small standard deviation indicating compact distributions for the prediction errors. As expected, queries that make use of more complex data structures (e.g., *flows*, *pattern search* and *top destinations*) incur in the larger errors, but still at most around 3% on average.

It is also very interesting to look at the features that the selection algorithm identifies as most relevant for each query. Remember that the selection algorithm has no information about what computations the queries perform nor what type of packet traces they are processing. The selected features give hints on what a query is actually doing and how it is implemented. For example, the number of bytes is the predominant traffic feature for the *pattern search* and *trace* queries when running on the trace with payloads. However, when processing the trace with just packet headers, the number of packets becomes the most relevant feature for these queries, as expected.

5.4 Prediction Cost

To understand the cost of running the prediction, we compare the CPU cycles of the prediction subsystem to those spent by the entire CoMo system over 5 executions. The feature extraction phase constitutes the bulk of the processing cost, with an overhead of 9.07%. The overhead introduced by the feature selection algorithm is only around 1.70% and the MLR imposes an even lower overhead (0.20%), mainly due to the fact that, when using the FCBF, the number of predictors is significantly reduced and thus there is a smaller number of variables to estimate. The use of the FCBF allows to increase the number of features without affecting the cost of the MLR. Finally, the total overhead imposed by our prediction method is 10.97%

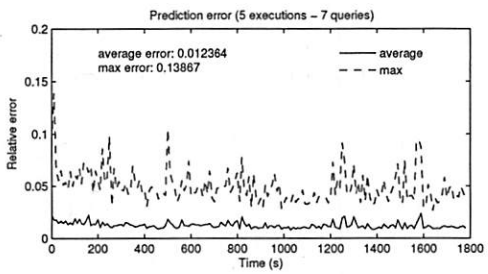


Figure 6: Prediction error over time (trace with payloads)

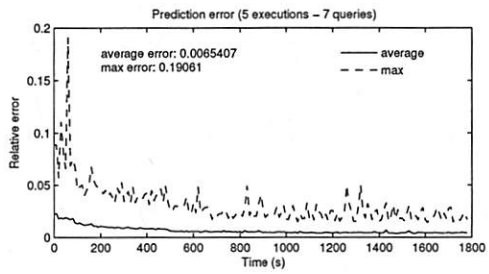


Figure 7: Prediction error over time (trace w/o payloads)

Trace with payloads

Query	Mean	Stdev	Selected features
application	0.0110	0.0095	packets, bytes
flows	0.0319	0.0302	*new* dst-ip, dst-port, proto
high-watermark	0.0064	0.0077	packets
link-count	0.0048	0.0066	packets
pattern search	0.0198	0.0169	bytes
top destinations	0.0169	0.0267	packets
trace	0.0090	0.0137	bytes, packets

Trace without payloads

Query	Mean	Stdev	Selected features
application	0.0068	0.0060	*repeated* 5-tuple, packets
flows	0.0252	0.0203	*new* dst-ip, dst-port, proto
high-watermark	0.0059	0.0063	packets
link-count	0.0046	0.0053	packets
pattern search	0.0098	0.0093	packets
top destinations	0.0136	0.0183	*new* 5-tuple, packets
trace	0.0092	0.0132	packets

Table 4: Breakdown of prediction error and selected features by query (5 executions)

6 Load Shedding

In this section, we provide the answers to the three fundamental questions any load shedding scheme needs to address: (i) when to shed load (i.e., which batch), (ii) where to shed load (i.e., which query) and (iii) how much load to shed (e.g., the sampling rate to apply). Algorithm 1 presents our load shedding scheme in detail, which controls the Prediction and Load Shedding subsystem of Figure 2. It is executed at each time bin (i.e., $0.1s$ in our current implementation) right after every batch arrival, as described in Section 3.1. This way, the system can quickly adapt to changes in the traffic patterns by selecting a different set of features if the current prediction model becomes obsolete.

6.1 When to Shed Load

To decide when to shed load the system maintains a threshold (*avail_cycles*) that accounts for the amount of cycles available in a time bin to process the queries. Since batch arrivals are periodic, this threshold can be dynamically computed as $(time\ bin \times CPU\,frequency) - overhead$, where $overhead$ stands for the cycles needed by our prediction subsystem (*ps_cycles*), plus those spent by other CoMo tasks (*como_cycles*), but not directly related to query processing (e.g., packet collection, disk and memory management). The overhead is measured using the TSC, as described in Section 4.4. When the predicted cycles for all queries (*pred_cycles*) exceed the *avail_cycles* threshold, excess load needs to be shed.

We observed that, for certain time bins, $como_cycles$ is greater than the available cycles, due to CoMo implementation issues (i.e., other CoMo tasks can occasionally consume all available cycles). This would force the system to discard entire batches, impacting on the accuracy of the prediction and query results. However, this situation can be minimized due to the presence of buffers (e.g., in the capture devices) that allow the system to use more cycles than those available in a single time bin. That is, the system can be delayed in respect to real-time operation as long as it is stable in the steady state.

We use an algorithm, inspired by TCP slow-start, to dynamically discover by how much the system can safely (i.e., without loss) exceed the *avail_cycles* threshold. The algorithm continuously monitors the system delay ($delay$), defined as the difference between the cycles actually used and those available in a time bin, and maintains a threshold ($rtthresh$) that controls the amount of cycles the system can be delayed without loss. $rtthresh$ is initially set to zero and increases whenever queries use less cycles than available. If at some point the occupation of the buffers exceeds a predefined value (i.e., the system is turning unstable), $rtthresh$ is reset to zero, and a second threshold (initialized to ∞) is set to $\frac{rtthresh}{2}$. $rtthresh$ grows exponentially while below this threshold, and linearly once it is exceeded.

This technique has two main advantages. First, it is able to operate without explicit knowledge of the maximum rate of the input streams. Second, it allows the system to quickly react to changes in the traffic.

Algorithm 1 (line 7) shows how the *avail_cycles* threshold is modified to consider the presence of buffers.

Algorithm 1: Load shedding algorithm

Input: Q: Set of q_i queries
b_i: Batch to be processed by q_i after filtering
como_cycles: CoMo overhead cycles
rtthresh, *delay*: Buffer discovery parameters

```
1  srate = 1;
2  pred_cycles = 0;
3  foreach q_i in Q do
4  |   f_i = feature_extraction(b_i);
5  |   s_i = feature_selection(f_i, h_i);
6  |   pred_cycles += mlr(f_i, s_i, h_i);
7  avail_cycles = (time_bin × CPU_frequency) -
   (como_cycles + ps_cycles) + (rtthresh - delay);
8  if avail_cycles < pred_cycles × (1 + error^) then
9  |   srate = max(0, avail_cycles − ls_cycles^) / (pred_cycles × (1 + error^));
10 |   foreach q_i in Q do
11 |   |   b_i = sampling(b_i, q_i, srate);
12 |   |   f_i = feature_extraction(b_i);
13 |   ls_cycles^ = α × Σ_i ls_cycles_i + (1 − α) × ls_cycles^;
14 foreach q_i in Q do
15 |   query_cycles_i = run_query(b_i, q_i, srate);
16 |   h_i = update_mlr_history(h_i, f_i,
   |   query_cycles_i);
17 error^ = α × |1 − pred_cycles / Σ_i query_cycles_i| + (1 − α) × error^;
```

Note that, at this point, *delay* is never less than zero, because if the system used less cycles than the available in a previous time bin, they would be lost anyway waiting for the next batch to become available.

Finally, as we further discuss in Section 6.3, we multiply the *pred_cycles* by $1+\widehat{error}$ in line 8, as a safeguard against prediction errors, where $\widehat{error}$ is an Exponential Weighted Moving Average (EWMA) of the actual prediction error measured in previous time bins (computed as shown in line 17).

6.2 Where and How to Shed Load

Our approach to shed excess load consists of adaptively reducing the volume of data to be processed by the queries (i.e., the size of the batch).

There are several data reduction techniques that can be used for this purpose (e.g., filtering, aggregation and sampling). In our current implementation, we support uniform packet and flow sampling, and let each query select at configuration time the option that yields the best results. In case of overload, the same sampling rate is applied to all queries (line 11).

In order to efficiently implement flow sampling, we use a hash-based technique called *Flowwise sampling* [11]. This technique randomly samples entire flows without caching the flow keys, which reduces significantly the processing and memory requirements during the sampling process. To avoid bias in the selection and deliberate sampling evasion, we randomly generate a new *H3 hash function* [6] per query every measurement interval, which distributes the flows uniformingly and unpredictably. The hash function is applied on a packet basis and maps the 5-tuple flow ID to a value distributed in the range [0, 1). A packet is then selected only if its hash value is less or equal to the sampling rate.

Note that our current implementation based on traffic sampling has two main limitations. First, using an overall sampling rate for all queries does not differentiate among them. Hence, we are currently investigating the use of different sampling rates for different queries according to per-query utility functions in order to maximize the overall utility of the system, as proposed in [23]. Second, there is a large set of imaginable queries that are not able to correctly estimate their unsampled output from sampled streams. For those queries, we plan to support many different load shedding mechanisms, such as computing lightweight summaries of the input data streams [22] and more robust flow sampling techniques [12].

6.3 How Much Load to Shed

The amount of load to be shed is determined by the maximum sampling rate that keeps the CPU usage below the *avail_cycles* threshold.

Since the system does not differentiate among queries, the sampling rate could be simply set to the ratio $\frac{avail_cycles}{pred_cycles}$ in all queries. This assumes that their CPU usage is proportional to the size of the batch (in packets or flows, depending on whether packet or flow sampling is used). However, the cost of a query can actually depend on several traffic features, or even on a feature different from the number of packets or flows. In addition, there is no guarantee of keeping the CPU usage below the *avail_cycles* threshold, due to the error introduced by the prediction subsystem.

We deal with these limitations by maintaining an EWMA of the prediction error (line 17) and correcting the sampling rate accordingly (line 9). Moreover, we have to take into account the extra cycles that will be needed by the load shedding subsystem (ls_cycles), namely the sampling procedure (line 11) and the feature extraction (line 12), which must be repeated after sampling in order to correctly update the MLR history. Hence, we also maintain an EWMA of the cycles spent in previous time bins by the load shedding subsystem (line 13) and subtract this value from *avail_cycles*.

After applying the mentioned changes, the sampling

Execution	Date	Time	Link load (Mbps) mean/max/min
predictive	24/Oct/06	9am:5pm	750.4/973.6/129.0
original	25/Oct/06	9am:5pm	719.9/967.5/218.0
reactive	05/Dec/06	9am:5pm	403.3/771.6/131.0

Table 5: Characteristics of the network traffic during the evaluation of each load shedding method

rate is computed as shown in Algorithm 1 (line 9). The EWMA weight α is set to 0.9 in order to quickly react to changes. It is also important to note that if the prediction error had a zero mean, we could remove it from lines 8 and 9, because buffers should be able to absorb such error. However, there is no guarantee of having a mean of zero in the short term.

7 Evaluation and Operational Results

We evaluate our load shedding system in a research ISP network, where the traffic load and query requirements exceed by far the capacity of the monitoring system. We also assess the impact of sampling on the accuracy of the queries, and compare the results of our predictive scheme to two alternative systems. Finally, we present the overhead introduced by the load shedding procedure and discuss possible alternatives to reduce it further.

7.1 Testbed Scenario

Our testbed equipment consists of two PCs with an Intel® Pentium™ 4 running at 3 GHz, both equipped with an Endace® DAG 4.3GE card [13]. Through a pair of optical splitters, both computers receive an exact copy of the link described in Section 5.1, which connects the Catalan Research and Education Network to the Internet. The first PC is used to run the CoMo monitoring system on-line, while the second one collects a packet-level trace (without loss), which is used as our reference to verify the accuracy of the results.

Throughout the evaluation, we present the results of three 8 hours-long executions (see Table 5 for details). In the first one (*predictive*), we run a modified version of CoMo that implements our load shedding scheme[5], while in the other two executions we repeat the same experiment, but using a version of CoMo that implements two alternative load shedding approaches described below. The duration of the executions was constrained by the amount of storage space available to collect the packet-level traces (600 GB) and the size of the DAG buffer was configured to 256 MB.

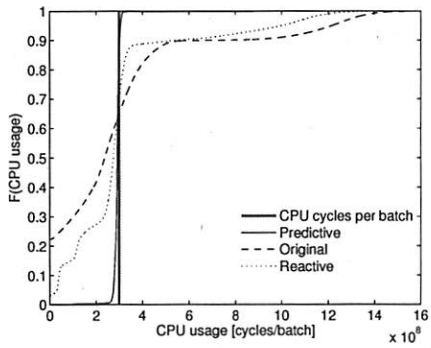


Figure 8: Cumulative Distribution Function of the CPU usage per batch

7.2 Alternative Approaches

The first alternative (*original*) consists of the current version of CoMo, which discards packets from the input buffers in the presence of overload. In our case, overload situations are detected when the occupation of the capture buffers exceeds a pre-defined threshold.

For the second alternative (*reactive*), we implemented a more complex reactive method that makes use of packet and flow sampling. This system is equivalent to a predictive one, where the prediction for a time bin is always equal to the cycles used to process the previous batch. This strategy is similar to the one used in SEDA [24]. In particular, we measure the cycles available in each time bin, as described in Section 6.1, and when the cycles actually used to process a batch exceed this limit, sampling is applied to the next time bin. The sampling rate for the time bin t is computed as:

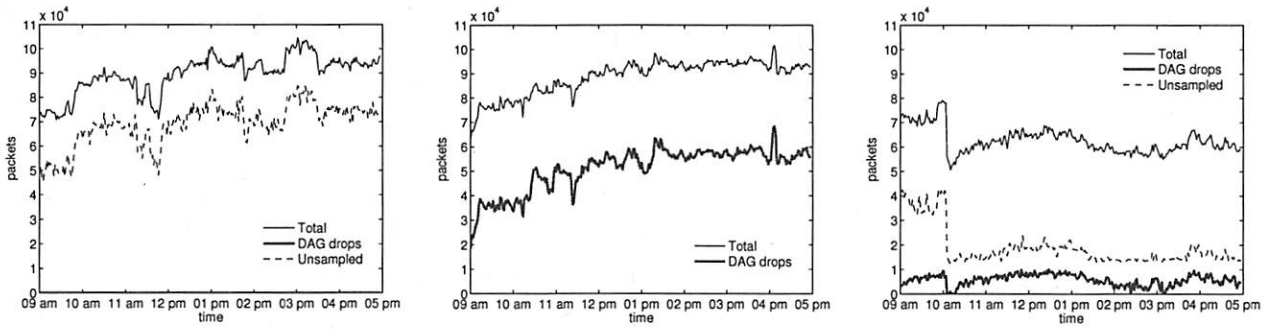
$$srate_t = \min\left(1, \max\left(\alpha, srate_{t-1} \times \frac{avail_cycles_t - delay}{consumed_cycles_{t-1}}\right)\right) \quad (3)$$

where $consumed_cycles_{t-1}$ stands for the cycles used in the previous time bin, $delay$ is the amount of cycles by which $avail_cycles_{t-1}$ was exceeded, and α is the minimum sampling rate we want to apply.

7.3 Performance

In Figure 8, we plot the Cumulative Distribution Function (CDF) of the CPU cycles consumed to process a single batch (i.e., the service time per batch). Recall that batches represent $100ms$ resulting in 3×10^8 cycles available to process each batch.

The figure shows that the *predictive* system is stable. As expected, sometimes the limit of available cycles is slightly exceeded owing to the buffer discovery algorithm presented in Section 6.1. The CDF also indicates good CPU usage between 2.5 and 3×10^8 cycles per batch, i.e., the system is rarely under- or over-sampling.



(a) *Predictive* load shedding (b) *Original* CoMo (c) *Reactive* load shedding

Figure 9: Link load and packet drops during the evaluation of each load shedding method

On the contrary, the service time per batch when using the *original* and *reactive* approaches is much more variable. It is often significantly larger than the batch interarrival time, with a probability of exceeding the available cycles greater than 30% in both executions. This leads to very unstable systems that introduce packet drops without control, even of entire batches. Figure 8 shows that more than 20% of the batches in the *original* execution, and around 5% in the *reactive* one, are completely lost (i.e., service time equal to zero).

Figure 9 illustrates the impact of exceeding the available cycles on the input stream. The line labeled 'DAG drops' refers to the packets dropped on the network capture card due to full memory buffers (results are averaged over one second). These drops are uncontrolled and contribute most to the errors in the query results. The line 'unsampled' counts the packets that are not processed due to packet or flow sampling.

Figure 9(a) confirms that, during the 8 hours, not a single packet was lost by the capture card when using the predictive approach. This result indicates that the system is robust against overload.

Figures 9(b) and 9(c) show instead that the capture card drops packets consistently during the entire execution[6]. The number of drops in the *original* approach is expected given that the load shedding scheme is based on dropping packets on the input interface. In the case of the reactive approach instead, the drops are due to incorrect estimation of the cycles needed to process each batch. The *reactive* system bases its estimation on the previous batch only. In addition, it must be noted that traffic conditions in the *reactive* execution were much less adverse, with almost half of traffic load, than in the other two executions (see Table 5). It is also interesting to note that when the traffic conditions are similar in all executions (from 9am to 10am), the number of unsampled packets plus the packets dropped by the *reactive* system is very similar to the number of unsampled packets by the *predictive* one, in spite of that they incur different processing overheads.

7.4 Accuracy

We modified the source code of five of the seven queries presented in Table 1, in order to allow them to estimate their unsampled output when load shedding is performed. This modification was simply done by multiplying the metrics they compute by the inverse of the sampling rate being applied to each batch.

We did not modify the *pattern search* and *trace* queries, because no standard procedure exists to recover their unsampled output from sampled streams and to measure their error. In this case, the error should be measured in terms of the application that uses the output of these two queries. As discussed in Section 6.2, we also plan to support other load shedding mechanisms for those queries that are not robust against sampling.

In the case of the *link-count*, *flows* and *high-watermark* queries, we measure the relative error in the number of packets and bytes, flows, and in the high-watermark value, respectively. The error of the *application* query is measured as a weighted average of the relative error in the number of packets and bytes across all applications. The relative error is defined as $|1 - \frac{estimated\ value}{actual\ value}|$, where the actual value is obtained from the complete packet trace, and all queries use packet sampling as load shedding mechanism, with the exception of the *flows* query that uses flow sampling.

In order to measure the error of the *top destinations* query, we use the detection performance metric proposed in [2], which is defined as the number of misranked flow pairs, where the first element of a pair is in the top-10 list returned by the query and the second one is outside the actual top-10 list. In this case, we selected packet sampling as load shedding mechanism [2].

Table 6 presents the error in the results of these five

Query	*predictive*	*original*	*reactive*
application (pkts)	1.03% ±0.65	55.38% ±11.80	10.61% ±7.78
application (bytes)	1.17% ±0.76	55.39% ±11.80	11.90% ±8.22
flows	2.88% ±3.34	38.48% ±902.13	12.46% ±7.28
high-watermark	2.19% ±2.30	8.68% ±8.13	8.94% ±9.46
link-count (pkts)	0.54% ±0.50	55.03% ±11.45	9.71% ±8.41
link-count (bytes)	0.66% ±0.60	55.06% ±11.45	10.24% ±8.39
top destinations	1.41 ±3.32	21.63 ±31.94	41.86 ±44.64

Table 6: Breakdown of the accuracy error of the different load shedding methods by query ($mean \pm stdev$)

queries averaged across all the measurement intervals. We can observe that, although our load shedding system introduces a certain overhead, the error is kept significantly low compared to the two reactive versions of the CoMo system. Recall that the traffic load in the $reactive$ execution was almost half of that in the other two executions. Large standard deviation values are due to long periods of consecutive packet drops in the alternative systems. It is also worth noting that the error of the *top destinations* query obtained in the *predictive* execution is consistent with that of [2].

7.5 Overhead

Figure 10 presents the CPU usage during the $predictive$ execution, broken down by the three main tasks presented in Section 6 (i.e., $como_cycles$, $query_cycles$ and $ps_cycles + ls_cycles$). We also plot the cycles the system estimates as needed to process all incoming traffic (i.e., $pred_cycles$). From the figure, it is clear that the system is under severe stress because, during almost all the execution, it needs more than twice the cycles available to run our seven queries without loss.

The overhead introduced by our load shedding system ($ps_cycles + ls_cycles$) to the normal operation of the entire CoMo system is reasonably low compared to the advantages of keeping the CPU usage and the accuracy of the results well under control. Note that in Section 5.4 the cost of the prediction subsystem is measured without performing load shedding. This resulted in an overall processing cost similar to the $pred_cycles$ in Figure 10 and therefore in a lower relative overhead.

While the overhead incurred by the load shedding mechanism itself (ls_cycles) is similar in any load shedding approach, independently of whether it is predictive or reactive, the overhead incurred by the prediction subsystem (ps_cycles) is particular to our predictive approach. As discussed in Section 5.4, the bulk of the prediction cost corresponds to the feature extraction phase, which is entirely implemented using a family of memory-efficient algorithms that could be directly built in hardware [15]. Alternatively, this overhead could be reduced significantly by applying sampling in this phase

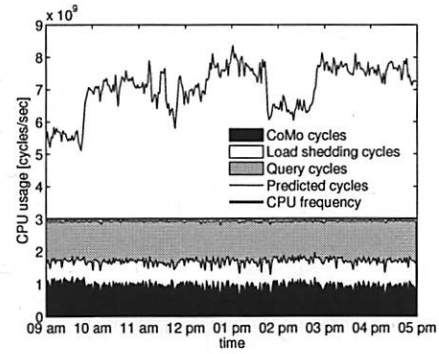


Figure 10: CPU usage after load shedding (stacked) and estimated CPU usage ($predictive$ execution)

or simply reducing the accuracy of the bitmap counters.

Finally, our current implementation incurs additional overhead, since it is not completely integrated with the rest of the CoMo system to minimize the number of modifications in the core platform. An alternative would be to merge the filtering process with the prediction in order to avoid scanning each packet twice (first to apply the filter and then to extract the features) and to share computations between queries that share the same filter rule. Better integration of the prediction and load shedding subsystem with the rest of the CoMo platform is part of our on-going work.

8 Conclusions and Future work

Effective load shedding methods are now indispensable to allow network monitoring systems to sustain the rapidly increasing data rates, number of users and complexity of traffic analysis methods.

In this paper, we presented the design and evaluation of a system that is able to predict the resource requirements of arbitrary and continuous traffic queries, without having any explicit knowledge of the computations they perform. Our method is based on extracting a set of features from the traffic streams to build an on-line prediction model of the query resource requirements, which is used to anticipate overload situations and effectively control the overall system CPU usage, with minimum impact on the accuracy of the results.

We implemented our prediction and load shedding scheme in an existing network monitoring system and deployed it in a research ISP network. Our results show that the system is able to predict the resources required to run a representative set of queries with small errors. As a consequence, our load shedding scheme can effectively handle overload situations, without packet loss, even during long-lived executions where the monitoring system is under severe stress. We also pointed out a sig-

nificant gain in the accuracy of the results compared to two versions of the same monitoring system that use a non-predictive load shedding approach instead.

In the paper, we have already identified several areas of future work. In particular, we are currently working on adding other load shedding mechanisms to our system (e.g., lightweight summaries) for those queries that are not robust against sampling. We also intend to develop smarter load shedding strategies that allow the system to maximize its overall utility according to utility functions defined by each query. Finally, we are interested in applying similar techniques to other system resources such as memory, disk bandwidth or storage space.

9 Acknowledgments

This work was funded by a University Research Grant awarded by the Intel Research Council, and by the Spanish Ministry of Education (MEC) under contract TEC2005-08051-C03-01 (CATARO project). Authors would also like to thank the Supercomputing Center of Catalonia (CESCA) for allowing them to collect the packet traces used in this work.

References

[1] BAR-YOSSEF, Z., JAYRAM, T. S., KUMAR, R., SIVAKUMAR, D., AND TREVISAN, L. Counting distinct elements in a data stream. In *Proc. of Intl. Workshop on Randomization and Approximation Techniques* (2002), pp. 1–10.

[2] BARAKAT, C., IANNACCONE, G., AND DIOT, C. Ranking flows from sampled traffic. In *Proc. of CoNEXT* (2005), pp. 188–199.

[3] BARLET-ROS, P., IANNACCONE, G., SANJUÀS-CUXART, J., AMORES-LÓPEZ, D., AND SOLÉ-PARETA, J. Predicting resource usage of arbitrary network traffic queries. Tech. rep., Technical University of Catalonia, 2006. http://loadshedding.ccaba.upc.edu/prediction.pdf.

[4] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (1970), 422–426.

[5] CARNEY, D., ET AL. Monitoring streams - a new class of data management applications. In *Proc. of Intl. Conf. on Very Large Data Bases* (2002), pp. 215–226.

[6] CARTER, J. L., AND WEGMAN, M. N. Universal classes of hash functions. *Journal of Computer and System Sciences 18*, 2 (1979), 143–154.

[7] CHANDRASEKARAN, S., ET AL. TelegraphCQ: Continuous dataflow processing of an uncertain world. In *Proc. of Conf. on Innovative Data Systems Research* (2003).

[8] CHI, Y., YU, P. S., WANG, H., AND MUNTZ, R. R. Loadstar: A load shedding scheme for classifying data streams. In *Proc. of SIAM Intl. Conf. on Data Mining* (2005).

[9] CISCO SYSTEMS. NetFlow services and applications. White Paper, 2000.

[10] DILLON, W. R., AND GOLDSTEIN, M. *Multivariate Analysis: Methods and Applications.* John Wiley and Sons, 1984.

[11] DUFFIELD, N. Sampling for passive internet measurement: A review. *Statistical Science 19*, 3 (2004), 472–498.

[12] DUFFIELD, N., LUND, C., AND THORUP, M. Flow sampling under hard resource constraints. In *Proc. of ACM Sigmetrics* (2004), pp. 85–96.

[13] ENDACE. http://www.endace.com.

[14] ESTAN, C., KEYS, K., MOORE, D., AND VARGHESE, G. Building a better NetFlow. In *Proc. of ACM Sigcomm* (2004), pp. 245–256.

[15] ESTAN, C., VARGHESE, G., AND FISK, M. Bitmap algorithms for counting active flows on high speed links. In *Proc. of ACM Sigcomm Conf. on Internet Measurement* (2003), pp. 153–166.

[16] IANNACCONE, G. Fast prototyping of network data mining applications. In *Proc. of Passive and Active Measurement* (2006).

[17] INTEL CORPORATION. *The IA-32 Intel Architecture Software Developer's Manual, Volume 3B: System Programming Guide, Part 2.* 2006.

[18] KEYS, K., MOORE, D., AND ESTAN, C. A robust system for accurate real-time summaries of internet traffic. In *Proc. of ACM Sigmetrics* (2005), pp. 85–96.

[19] L'ANELLA CIENTÍFICA (THE SCIENTIFIC RING). http://www.cesca.es/en/comunicacions/anella.html.

[20] NLANR: NATIONAL LABORATORY FOR APPLIED NETWORK RESEARCH. http://www.nlanr.net.

[21] PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. 1992.

[22] REISS, F., AND HELLERSTEIN, J. M. Declarative network monitoring with an underprovisioned query processor. In *Proc. of Intl. Conf. on Data Engineering* (2006), pp. 56–67.

[23] TATBUL, N., ET AL. Load shedding in a data stream manager. In *Proc. of Intl. Conf. on Very Large Data Bases* (2003), pp. 309–320.

[24] WELSH, M., CULLER, D. E., AND BREWER, E. A. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of ACM Symposium on Operating System Principles* (2001), pp. 230–243.

[25] WHANG, K.-Y., VANDER-ZANDEN, B. T., AND TAYLOR, H. M. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst. 15*, 2 (1990), 208–229.

[26] YU, L., AND LIU, H. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *Proc. of Intl. Conf. on Machine Learning* (2003), pp. 856–863.

Notes

[1] The choice of $100ms$ is somewhat arbitrary. Our experimental results indicate that $100ms$ represents a good trade-off between accuracy and delay with the traces of our dataset. We leave the investigation on the proper batch duration for future work.

[2] A description of the queries used in our experiments can be found in [3]. The actual source code of all queries is also publicly available at http://como.sourceforge.net.

[3] It is possible that the CPU usage of other queries may exhibit a non-linear relationship with the traffic features. A possible solution in that case is to define new features computed as non-linear combinations of simple features.

[4] Note that the values of some predictors may become very similar under special traffic patterns. For example, the number of packets and flows can be highly correlated under a SYN-flood attack.

[5] The source code of the prediction and load shedding system is available at http://loadshedding.ccaba.upc.edu. The CoMo monitoring system is also available at http://como.sourceforge.net.

[6] The values are a lower bound of the actual drops, because the loss counter present in the DAG records is only 16-bit long.

Configuration Management at Massive Scale: System Design and Experience

William Enck, Patrick McDaniel
Pennsylvania State University
{enck,mcdaniel}@cse.psu.edu

Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel
AT&T Research
{sen,psebos}@research.att.com
sspoerel@att.com

Albert Greenberg
Microsoft Research
albert@microsoft.com

Sanjay Rao
Purdue University
sanjay@ecn.purdue.edu

William Aiello
University of British Columbia
aiello@cs.ubc.ca

Abstract

The development and maintenance of network device configurations is one of the central challenges faced by large network providers. Current network management systems fail to meet this challenge primarily because of their inability to adapt to rapidly evolving customer and provider-network needs, and because of mismatches between the conceptual models of the tools and the services they must support. In this paper, we present the PRESTO configuration management system that attempts to address these failings in a comprehensive and flexible way. Developed for and deployed over the last 4 years within a large ISP network, PRESTO constructs device-native configurations based on the composition of *configlets* representing different services or service options. Configlets are compiled by extracting and manipulating data from external systems as directed by the PRESTO configuration scripting and template language. We outline the configuration management needs of large-scale network providers, introduce the PRESTO system and configuration language, and demonstrate the use, workflows, and ultimately the platform's flexibility via an example of VPN service. We conclude by considering future work and reflect on the operators' experiences with PRESTO.

1 Introduction

Configuration management is among the largest cost-drivers in provider networks. Such costs are driven by the immense complexity of the supported services and infrastructure. For a large provider, thousands of enterprises with diverse services and configurations must be seamlessly and reliably connected across huge geographic areas and rapidly evolving networks. Moreover, the initial turn-up installation and subsequent support of a single customer may span many organizations and systems internal to the provider. The stakes for the supported enterprise are extremely high: an outage may result in loss of business, delays in "getting to revenue" since service turn up precedes revenue, failure to meet contractual obligations, or disruption of key organizational workflows.

Given the complexity and stakes, it may be surprising that the common configuration management practice involves pervasive manual work or ad hoc scripting. The reasons for this are multi-faceted. From a provider perspective, every customer is in some ways unique. Though service orders have a lot in common, many new installations made to realize those orders represent unique combinations of services and network configurations. Interactions with customer networks, stale, incomplete, or imperfect information, and service interactions make both turn-up as well as ongoing maintenance of configurations complex and error-prone processes. Moreover, the devices in the network and definition of services they support change at a dizzying rate. New firmware versions, customer requirements, or supported applications appear every day. Market demands further dictate that the time-to-market for new services is a critical driver of revenue: delays caused by tool configuration, extension, or development can mean the difference between profitability and loss. In short, there is an unserved need in provider networks for tools that address these complex and sometimes contradictory challenges while constructing service configurations.

The PRESTO configuration management system develops network device configurations from composed collections of *configlets* that define the services to be supported by the target device. Written in our general-purpose hybrid scripting and configuration template language, the PRESTO system extracts specific information from external systems and databases and transforms this information into complete device configurations as directed by the PRESTO compiler. Extensive interactions with diverse engineering teams charged with managing operational IP networks led us to the conclusion that, to gain wide buy-in and adoption, the PRESTO language must adhere closely to the complex and often

low level configuration languages supported by network device vendors (e.g., for Cisco devices, the IOS command language). PRESTO empowers network designers, "power users" comfortable with native network device configuration languages, and automates the unambiguous translation of their design rules into precise network configurations. Specifically, the PRESTO system generates complete device-native configurations, which can then be downloaded into a device and archived by network operators.

In this paper, we present the motivation, design, and workflow of the the PRESTO configuration management system. We outline the challenges faced by a large network provider in installing and maintaining millions of diverse devices for thousands of customers and organizations, and reflect on the failures of past network management systems to address these needs. We outline the PRESTO workflow and configuration language and demonstrate its features through an extended example. Finally, we discuss future work and detail preliminary experiences in deploying services in operational networks.

To date, PRESTO has concentrated on developing "greenfield" configs–configuration of new routers or services in a new installation. Such an approach avoids the inherent complexity of dealing with post-turn-up manipulation of fielded configurations resulting from software updates, performance tuning, or other maintenance. PRESTO's role in the long-term maintenance of routers, called "brownfield"configuration, is currently evolving. While the body of the following work focuses on greenfield use, we revisit this latter objective and the challenges therein in our concluding remarks.

The PRESTO system evolved out of decades of experience in network management. Configuration management is about more than just getting routing and filtering correct. It must meld together many different services that exhibit subtle interactions and dependencies. Therein lies the challenge of configuration management in a provider network—*How do we glue together many information sources of myriad organizations in real time to build a functioning device configuration?* Revisited in the following section and throughout, it is the lessons gleaned from our experiences in meeting that goal that drive the PRESTO design.

The remainder of the paper proceeds as follows: Section 2 discusses motivation and requirements; Section 3 overviews the PRESTO work flow; Section 4 describes the language extensions provided by PRESTO; Section 5 incorporates these language extensions into a usable system, recognizing that input data is rarely pristine; Section 6 describes our experience using PRESTO for a real application within an enterprise network; Section 7 discusses related work; and Section 8 concludes.

2 Configuration Automation

In this section, we discuss the need for automation by describing current best practices and their limitations. We then describe the challenges an automated configuration generation system must face in large provider networks.

A router configuration file[1] provides the detailed specification of the router's configuration, which in turn determines the router's behavior. In essence, the configuration file is a representation of the sequence of specific commands that if typed through the command line interface determine the wide set of interdependent options in router hardware and software configuration. In practice, this may represent thousands of lines of complex commands, per router. These configuration files are text artifacts – described in a device specific command language, with a device specific syntax in a human and machine readable format, in some cases in XML. It is worth noting that a plethora of network devices beyond routers, e.g. Ethernet switches and firewalls, rely on configuration files of this type. To some degree, this state of affairs reflects natural technology evolution and the marketplace – networks started (and often still start) small and therefore often gravitated toward manual or (ad hoc) scripted configuration.

Today's configuration languages offer myriad complex options, typically described in precise low level device-specific languages, such as Cisco's IOS command language [8]. While the learning curve for such languages might be steep and the cost of inadequate learning severe (small slipups may cause large network outages), these languages are in extremely wide use for the entire lifecycle of network management – starting with configuration, but encompassing all other aspects, including performance, fault and security management. The tack that the PRESTO system takes is to leverage these "native" languages, and empower the user of these languages to enforce the precise translation of design intent into detailed device configuration.

Our interactions with network designers revealed that using templates to describe design intent is essentially universal. That is, designers create parameterized chunks of design configurations to describe intent. Accordingly, PRESTO provides full and flexible support for template creation in the native device configuration language.

2.1 Need for Automation

Decades of experience in network management have taught us that manual configuration practices are limited in the following ways, i.e., *the configuration process is*:

• *costly, time-consuming, and unscalable:* There is a significant initial investment in the interpretation and documentation of network standards and device-specific in-

terfaces in developing any new service or support for a device. The result of that investment is a "model" configuration document (sometimes termed an Engineering and Troubleshooting Guidelines (ETG) document) used by enablers[2] to manually configure each target device. Typically performed by a large network engineering organization and depending on the complexity of the service or device, this process can take many person months of effort to complete and is an expensive process. The subsequent manual application of the model configurations to customer networks is also costly—some large customers may have tens of thousands of network elements, and applying a new configuration to even a fraction of them verges on the intractable.

• *prone to misinterpretation and error:* Even under the best of circumstances, engineering guidelines will not be perfect. Because network designers cannot anticipate all possible target environments, the guidelines are necessarily ambiguous, sometimes imprecise, and often subject to multiple interpretations. Thus, different enablers may interpret the same rules differently or adopt different local conventions. Differences between interpretations can and often do result in configuration mismatch errors. Making matters worse, while some errors might be easier to detect, others might have no immediate effect. These latter configuration problems are the most vexing, as they may become manifest at periods of high load (possibly the worst possible time) or introduce undetected security vulnerabilities.

• *fraught with ambiguous, incorrect, changing or unavailable input data:* Configuration information is not only spread across multiple data sources, but may be incomplete and imperfect. For example, customer order databases may not reflect the latest needs of the customer, e.g., order updates may only exist as emails to human contacts and may not quickly (or ever) be reflected in a database. As another example, information such as IP address assignments may be missing at the time of initial configuration. Finally, rules might be ambiguous. We have, for example, encountered examples where a particular service mandated that a site may have dual routers with ISDN backup, but it was not obvious which router must be the backup and which the primary.

2.2 Requirements

The pervasive practices and technical organizational problems detailed above makes automation in provider networks difficult. These issues lead to the following requirements, i.e., *the configuration process must*:

• *Support existing configuration languages:* While there have been many prior strong efforts at automating configuration generation, most have focused on developing abstract languages or associated formalisms to specify configurations in a vendor neutral fashions, e.g., IETF standard SNMP MIBs (Simple Network Management Protocol, Management Information Bases) [5], and the Common Information Model (CIM) [9]. These information models define and organize configuration semantics for networking/computing equipment and services in a manner not bound to a particular manufacturer or implementation. However, such generalized abstractions invariably introduce layers of interpretation between the specification and device. Gaps open between general abstract specifications and the concrete language of specific device configuration. It is very difficult to avoid extending or creating specialized common models to describe the realities of today's rapidly evolving devices and services. The artifacts of efforts to create standards or libraries often lag the marketplace. In truth, network experts often do not have the time or inclination to understand such abstractions, and today nearly universally find that working within native configuration interfaces is much more efficient for initial installation and later maintenance.

• *Scale with options, combinations, and infrastructure:* Customer configurations are dependent on, in particular, selected service offerings, devices, firmware revisions, and local infrastructure. For example, consider a site connecting to a provider backbone. The seemingly simple customer order has many options—does the customer require multiple routers to connect to the backbone or just one? Should each have multiple links or one? Further, each router may have several WAN (Wide Area Network) and LAN (Local Area Network) facing interfaces, and each interface may admit specific naming conventions that depend on the router model and the WAN card. The physical local infrastructure (e.g., routers and network topologies), will often have major impact on the workflow and content of the configuration.

• *Support heterogeneous and diverse data sources:* Putting together a router configuration involves collecting all necessary router configuration information. Such configuration information may not be all available in one central repository at the time the information is needed, but rather maybe distributed amongst a variety of databases, which are populated by upstream workflows. Take, for example, the customer order database. The customer information may itself have arrived at different times, and may be split across various forms. In large operational networks, information regarding customer orders and the resulting router deployment and maintenance is potentially spread across various systems spanning many internal organizations. An automated configuration system has to be cognizant of the diversity of information sources (and quality of data). Importantly, these data sources typically have their own persis-

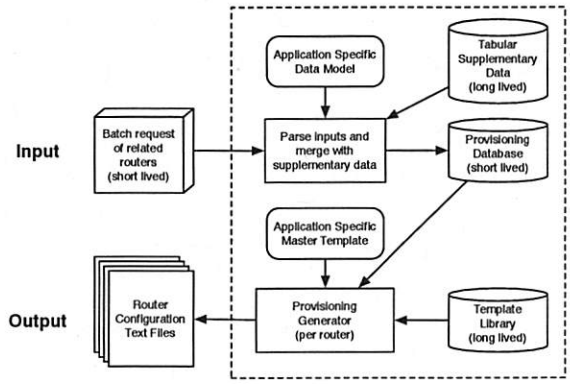


Figure 1: PRESTO Workflow

tent databases (each representing large investments), and a configuration management solution faces a huge real-world hurdle if it were to attempt to replace or replicate these databases, or even add a new persistent database rather than extend an already existing one. To the greatest extent possible, a configuration management system should strive to be stateless if it is to succeed in diverse operational environments.

3 PRESTO Overview

Two kinds of users interact with PRESTO—domain experts and provisioners. Domain experts initially codify the configuration services and options in *active templates*. These are the PRESTO equivalent of the ETG, where execution of the "active" template by the compiler directs the interpretation of the service definition for a particular environment. At installation time, provisioners obtain necessary configuration information from customers and other sources, that, when combined with the templates written by the domain expert, produce the end router configuration.

A more complete description of the PRESTO workflow is presented in Figure 1. In practice, deployment is a multi-stage process including requirements reconnaissance, initial staging configuration creation, and device turn-up. PRESTO is concerned with the second part of the process, the creation of the configuration. The configuration creation process begins with a batch of new configuration requests resulting from a network upgrade request or customer order. The requests are submitted to PRESTO as a collection of specification inputs, where the relevant environment data is provided as direct input or extracted from supplementary inventory and configuration databases. The data is cleansed and projected into a service or device (application-specific) data model created for the target configurations. Finally, an active template is executed for the set of specified target devices resulting in complete device-native configurations.

Note that these active templates, called *configlets*, may conditionally or deterministically import arbitrary other templates or extract data as needed at compile time to complete a configuration. It is this "composition by recursive import" and dynamic data extraction from external sources that allows PRESTO to deal with the potentially huge number of configuration options for a configuration.

PRESTO adopts automation to the extent possible, but allows for human intervention where needed. The process of mining and preparing configuration information is decoupled from the process of combining input data with configlets to produce configurations. Hence, as discussed further in Section 5.1, PRESTO can be viewed as a 2-step process, each of which is automated, but with manual intervention in between the steps. The first step involves tools that can, in an automated fashion, parse disparate databases to extract necessary configuration information. The information so produced may potentially be inspected by an enabler, overridden if necessary, and augmented with manually supplied missing information. The cleansed information is then the input to the second stage of the automated processing to produce the configuration.

4 Active Template Language

The PRESTO template language lays out the foundation over which the rest of the system is built. PRESTO does not define a new language, but rather a language *extension*. This allows domain experts to leverage knowledge of flexible native configuration languages, e.g., Cisco IOS, while creating useful abstractions, defining services, which take the form of *configlets* in PRESTO. Interestingly, this is in direct opposition to traditional network management interfaces which provide a single abstraction to which any policy would have to adhere. PRESTO provides the following key characteristics:

- *Data Modeling:* The data used to configure a network device is derived from a complex calculus of data from many diverse sources. Hence, dealing with this requires some means of organizing and accessing the data. We use the most natural choice for this task: a relational database. The schema of that database, called a *model*, is specific to the services and devices to which it applies, i.e., each collection of configlets works in tandem with a model defined by the domain experts.

- *Rich Template language:* A straightforward view of templates is that this merely involves direct substitution of "variables" by user-supplied inputs. For example, an architect may insert a variable for IP address information, that is then supplied by inputs from the user.

However, more complex operations and data manipulations are needed. For example, based on whether or not a router has a particular feature, the configuration may need to omit or include executable script code. Again, a device may have one or more interfaces, and configuration blocks may need to be created for each (of possibly many) interfaces. Moreover, the number of interfaces to be configured may only be available at run-time. In short, more sophisticated constructs than purely variable substitution are needed, e.g., variable expressions, conditionals, and loops.

• *Support for Template Decomposition and Assembly:* PRESTO provides support for the architect to write multiple smaller templates targeted for very specific elements of a configuration, i.e., services. There are several advantages to such an approach. First, supporting multiple smaller templates simplifies creation and maintenance. Second, this allows for templates to be written by multiple designers based on their expertise. This is analogous to programming modularity, where each programmer or group can be develop and maintain a part of the larger system. In the case of PRESTO, for example, a designer may better understand how to deal with various WAN interfaces, while another may better understand issues with BGP configuration. Third, breaking templates down promotes reusability, as there is the potential to create template libraries, and reuse them across multiple applications.

Before discussing specific details, we provide an example scenario to aid in the understanding of language constructs and design motivations. Consider the configuration of a gateway router. The gateway connection may have one or more external connections. If there are multiple connections, they may be dispersed across multiple routers. Hence, these routers require configuration knowledge of the other routers participating in the gateway connection, e.g., IP addresses, to coordinate failover, e.g., HSRP [17]. The template language must support these relationships between connections on one router and between routers.

We now discuss each part of the template language in turn. After discussing the core language concepts, we introduce additional language features that enable better software engineering practices.

4.1 Data Model

The PRESTO template language revolves around the data model. The templates require access to small data chunks describing router properties. Furthermore, multi-router relationships dictate a need to perform quick lookups for peer specific information. Such a capability is required for instance when configuring one router (eg., a spoke) involves extracting information for another router (eg., the hub). A relational database provides just this capability: router properties are stored in table fields and accessed as variables; peer router properties are queried by specifying the router hostname. Hence, the data model becomes a database schema. We refer to this database as the provisioning database and provisioning relational database schema where necessary to remove ambiguity.

The schema definition is application dependent. Each application has different requirements on data accessibility. It is no surprise that defining the schema is the most delicate part of applying PRESTO to a new application, hence complete flexibility is required. Despite the supported flexibility, past experience has resulted in a few recommended guidelines. The data model should contain a `ROUTER` table indexed by a globally unique identifying value, e.g., the router hostname. One row will exist for each router in a provisioning request. The `ROUTER` table should contain the bulk properties; however, whenever multiple instances of a property occur, a new table should be created. For example, multiple LAN or WAN interfaces are semantically equivalent. Sub-router tables should use a multi-column primary key consisting of the `ROUTER` table index and a unique identifier, e.g., interface number. Upon querying the database for all LAN interface records matching a specific router hostname, the template language produces an interface loop. For example; suppose `LAN` is a table holding all LAN-facing inputs. Then

```
SELECT * FROM LAN WHERE ROUTER=THIS_ROUTER
```

selects each LAN-facing interface on a given router. Iteration specifics and syntax are discussed below.

4.2 Variable Evaluation

Variable substitution is integral to any template language; PRESTO is no exception. Variables are defined by the data model. Templates gain access to variables by querying the provisioning database. The returned record defines a variable namespace, or *context*[3], used to access the variable, e.g.:

```
<CONTEXT.VARIABLE>
```

Variables of this form are directly substituted in the template text.

Templates are written to produce a configuration file for one router. PRESTO begins template evaluation by querying the `ROUTER` table in the data model for the row corresponding to the current router. The returned record populates the `ROUTER` context, which consequently allows templates to use `ROUTER` variables at any point. The template creates new contexts by making a new

database query; however, those variables are only accessible within the defined context scope. When a query returns multiple records, the template code within the context is repeated, producing a loop. For example, SELECT * FROM LAN WHERE ROUTER=THIS_ROUTER has the effect of configuring multiple LAN-facing interfaces.

4.3 Iteration

The PRESTO template language simulates iteration by executing database queries that return multiple records. The template designer creates a data driven loop by defining a new context name, an SQL-like query, and a scope. Each row returned by the query produces an iteration. For example:

```
[INT:SELECT (*) FROM (WAN_INTERFACE) WHERE
 (WAN_INTERFACE.HOSTNAME=<ROUTER.HOSTNAME>)]
interface serial0/<INT.SLOT>/<INT.PORT>
 bandwidth <INT.BANDWIDTH>
 ip address <INT.IP> <INT.MASK>
!
[/INT]
```

Here, `INT` is the name of the new context. The statement associates the `INT` context with the record returned by querying the `WAN_INTERFACE` table of the provisioning database for all fields (`(*)`) related to the current router hostname (note the use of `<ROUTER.HOSTNAME>` in the query). The text within the `INT` context scope, i.e., all text between the query statement and the context closing statement, `[/INT]`, is repeated for each returned record. Field names from each record are accessible as variables within the context, as shown. Note that new context definitions can be arbitrarily nested, but they cannot define scopes spanning multiple parent scopes. That is, the nested context's closing statement must occur before its parent closing statement. This constraint is consistent with loop structures in common programming languages.

4.4 Conditional Logic

Configuration statements are commonly dependent on router properties. For example, E1 (a standard widely used in Europe) line cards required slightly different interface specification than T1 (a standard widely used in the US) cards. The PRESTO template language supports the inclusion and omission of configuration options with conditional statements. All conditionals have a label, condition and scope, in general:

```
[COND LABEL CONDITION]
... template text
[/LABEL]
```

`COND` indicates a conditional statement; `LABEL` defines a label; and `CONDITION` contains relational operators that dictate if the template text between the condition statement and the closing statement, `[/LABEL]`, is included. The template text can contain static strings, new contexts, or even more conditionals. The `CONDITION` itself supports arbitrary complexity of Boolean logic. Statements can be simple:

```
("<ROUTER.HAS_FEATURE_X>" eq "YES")
```

or more complex logic:

```
(("<ROUTER.HAS_FEATURE_X>" eq "YES") &&
(("<ROUTER.HAS_FEATURE_Y>" eq "YES") ||
  ("<ROUTER.FEATURE_Z>" ne "BASIC")))
```

4.5 Data Transformation

Configuration statements commonly require a transformation of an input variable. For example, an interface IP address may be specified as IP and mask, i.e., *x.x.x.x/y*, but the router configuration language requires the IP and mask coded separately, i.e., *x.x.x.x z.z.z.z*. In another case, the template designer may need to configure the network address corresponding to the input value. To accommodate such requirements, the PRESTO template language provides a mechanism for arbitrary extension.

A function added to the language interpreter module is referenced within a template as a context, variable, function, and argument:

```
<CONTEXT.NEW_VARIABLE:function(args)>
```

Upon execution, arguments are evaluated (if they are variables) and passed to the function. The function performs a specific manipulation and returns the result to a new variable in the specified context. The new variable's value is inserted into the template text, and it's value is retained for later use within the context.

To aid template design, the PRESTO template language contains a core set of application agnostic functions. Some functions provide generic computation abilities, e.g., `calc()` performs simple arithmetic, `sbsstr()` returns a substring specified by an offset and length, and `matchre()` provides regular expression substring matching. Other core utility functions perform useful conversions on common network values such as IP address. For example:

```
<INT.NETIP:computeOffsetMaskIP(<INT.IP>,0)>
```

computes the network address of an IP specified in *x.x.x.x/y* form. The function, however, can calculate any offset of the IP, a useful feature when network policy dictates devices on specific offsets, e.g., the gateway is commonly `.1`. Realizing a new PRESTO function involves including its code in the PRESTO language interpreter.

4.6 Hidden Evaluation

Configuration policy occasionally requires values resulting from complex computations. While additional domain specific functions provide ample mechanism, template designers are encouraged to keep domain knowledge within the templates themselves. The motivation is twofold. First, this reduces bloat of the core language. Second, as function definitions require programming, and most template designers do not possess the necessary skills, or are simply unwilling, to create new functions. Therefore, we have added only a small number of generic primitive operations to the core language in an application.

As described to this point, the template language is not conducive to performing complex computation within the templates themselves. All functions return text that is inserted into the end router configuration. Multi-step computations therefore become difficult, if not impossible. To overcome this issue, the language supports hidden evaluation:

```
[EVAL LABEL noprint]
... template statements
[/LABEL]
```

Statements within the `LABEL` scope produce no output.

Computation within `EVAL` blocks is not limited to simple multi-step functional transformations. In practice, we leveraged the hidden evaluation interface to provided a multitude of features. For example, database `SELECT` queries were used to lookup values in supplemental data tables. Values were assigned to higher level contexts, e.g., `ROUTER`, and used throughout the template. The `EVAL` blocks also proved useful to determine values that depended on multiple conditionals. The conditional logic was performed once, and the value was used many times thereafter.

4.7 Template Assembly

Managing one large template becomes unwieldy. Software engineering experience recommends modular code. Templates are no exception. Using many small templates, or `configlets` provides many beneficial side effects. It allows a template designer to concentrate on one feature at a time. For example, a configlet can be written for each network access type used for the WAN interface of a router. Later, depending on the router provisioning data, the correct configlet is chosen. By including configlets on demand, complicated conditional logic is avoided. Additionally, as configlets are only inserted where applicable, they can be written with certain assumptions in mind. This reduces complexity within the configlets themselves. Finally, as configlets can include other configlets, the template designer can exploit commonality between configlets.

PRESTO stores all configlets in a template library. Configlets can be included at any point. The language provides a special syntax for including configlets:

```
[INCLUDE FROM (FEATURE) WHERE
  (FEATURE.TYPE=SOME_TYPE)]
```

In our above example, the correct WAN interface configlet is included using the `<ROUTER.ACCESS_TYPE>` variable:

```
[INCLUDE FROM (WAN) WHERE
  (WAN.ACCESS_TYPE=<ROUTER.ACCESS_TYPE>)]
```

4.8 Example Configlet

Once the data model and configlet organization are determined, writing the configlets themselves is straightforward. We now provide a quick example to show how each of the language primitives come together. Consider a network topology where the Internet edge has two access lines, each connected to one router, and the two routers establish load sharing of in and outbound traffic. The most complex configlet will define the WAN routing protocol. Figure 2 provides an example.

The example begins by including the interface configlet defining the back to back connection between the two routers. Multiple connection types may be supported. Instead of using a large conditional statement to pick the right interface definition, the `INCLUDE` statement allows the relational database to perform the conditional logic and simplify the code the domain expert must specify.

The BGP block defines the WAN routing protocol configuration. Here, a `SELECT` queries for network addresses specific to the peer router. The configlet also performs a domain specific sanity check that warns the enabler if the two routers' back to back IP address are in different networks. This check is placed within and `EVAL` block to keep warning text from leaking into the end configuration. Finally, the `SELECT` query is also used to determine information specific to the WAN connection. Here, the data model specifies that WAN interface specifics be placed in a separate table. The configlet selects the correct table row using the `HOSTNAME` foreign key. The Cisco IOS `network` and `neighbor` commands require a translation of the available information, therefore the `computeOffsetMaskIP()` function is used. In this case, the remote peer is always the second IP in the network, therefore the domain expert is able to code the neighbor's IP directly using the offset function.

```
%% Configure Back-to-Back Interface to Peer Router
[INCLUDE FROM (B2B) WHERE (B2B.TYPE=<ROUTER.B2B_TYPE>)]
%% Define the BGP configuration
router bgp <ROUTER.LOCAL_ASN>
 network <ROUTER.LOOPBACK0> mask 255.255.255.255
[PEER:SELECT FROM (ROUTER) WHERE (ROUTER.HOSTNAME=<ROUTER.PEER>)]
%% Ensure the peer is in the same network
[EVAL B2B_CHECK noprint]
[COND WRONGNET ("<ROUTER.B2BNET:computeOffsetMaskIP(<ROUTER.B2B_IP>,0)>" \
                ne "<PEER.B2BNET:computeOffsetMaskIP(<PEER.B2B_IP>,0)>" ) ]
<ROUTER.WARNING:templateWarning(<ROUTER.HOSTNAME> and <PEER.HOSTNAME> different B2B Net)>
[/WRONGNET]
[/B2B_CHECK]
 network <PEER.NETIP:computeOffsetMaskIP(<PEER.B2B_IP>,0)> mask 255.255.255.252
 neighbor <PEER.B2B_IP> remote-as <ROUTER.LOCAL_ASN>
 neighbor <PEER.B2B_IP> next-hop-self
[/PEER]
[WAN:SELECT FROM (WAN) WHERE (WAN.HOSTNAME=<ROUTER.HOSTNAME>)]
 network <WAN.NETIP:computeOffsetMaskIP(<WAN.IP>,0)> mask <WAN.MASK:computeMask(<WAN.IP>)>
%% The gateway is the second IP in the subnet (for this example)
 neighbor <WAN.GW:computeOffsetMaskIP(<WAN.IP>,1)> remote-as <WAN.REMOTE_ASN>
[/WAN]
 no auto-summary
!
```

Figure 2: Example configlet defining the WAN routing protocol of a two-line two-router configuration

5 The PRESTO System

The template language and data model provides a mechanism to describe configuration policy; however, it must be incorporated into a usable system. In an ideal world, an engineer receives a request for a group of related routers with all required input information available and correct. This would allow a straightforward application of the data model and templates to act upon inputs. As shown in Figure 1, the per-request input data is parsed and merged with tabular supplementary rules to create a one time database. Then, for each router in the request, a master active template is executed by the provisioning generator. This master template includes and executes appropriate configlets according to the input data, resulting in a completed configuration text file.

Unfortunately, the information required to configure a router is not always readily available. In large operational networks, the input data for the configuration task spans the outputs of multiple upstream workflows, which may arrive at different points in time. It is therefore important to be able to work with such partial information flows and to able to handle any inconsistencies across the flows. In such a scenario, ubiquitous flow-through or full automation becomes extremely difficult to realize. Accordingly, PRESTO provides hooks to overcome these difficulties, when and where they arise.

5.1 PRESTO Architecture

PRESTO achieves nearly full automation using a 2-step process, see Figure 3. The goal of automation is to minimize user actions. PRESTO minimizes manual processes in two ways. First, it handles bulk requests. This streamlines the process of creating the initial router configuration code. Second, it requires only one point of user interaction at which point users provide the most minimal effort to allow automation to complete. In PRESTO, data processing proceeds in two steps, with user integration capabilities made available between step 1 and step 2.

Specifically, PRESTO uses a 2-step architecture to request user input at the most ideal moment. The process begins with the submission of a batch request to step 1. Step 1 pulls together and parses information from available input data sources for the batch request. The output of step 1 is the complete and unambiguous information needed to configure all routers in the batch. The role of step 1 then is to normalize and tabulate the input information and, if possible, apply defaults or inference rules to fill in missing information. Where defaults or inference are applied, the step 1 output will flag or annotate the output, for (optional or mandatory) user inspection. In practice, we found that inference rules include many types of calculations, for example, ranging from assigning incremental BGP AS prepend values to selecting interface ports for network connections. The result is presented to the user for validation in tabular form. The user is then given the option to change certain of the data to meet customer requirements (which we found sometimes change faster than the ordering information systems can be updated), e.g., changing interface cards, and the batch request is submitted to step 2. Step 2 executes the PRESTO engine described in Figure 1, combining complete input information with templated policy information as described above, to produce the configuration file for each router in the batch request. By dividing the

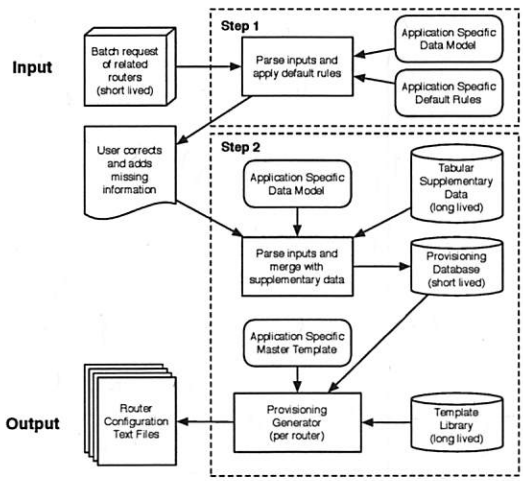


Figure 3: 2-step PRESTO Data Flow

processing in these two steps, we isolate fallout due to inadequate information to step 1, and provide the system and its users opportunities to repair fallout and resume automation before proceeding to step 2, which then produces the desired output.

PRESTO was designed to be input agnostic, and has been adapted to be invoked in a number of different modalities (via web services, via database invocations, or via stand-alone interfaces), and to operate on inputs of various forms and origins (database extracts, Excel spreadsheets, XML forms). PRESTO is essentially stateless. By design, the persistent data in PRESTO is limited to a a repository of configuration policies; the persistent database of associated router configurations is external to PRESTO.

Accordingly, PRESTO applications are built on a *provisioning data model*, describing short-lived data, and a *policy data model*, describing long lived data (see Figure 3). We do not have space here to describe the details or the precise representations of these data models, and so shall describe salient features. Short lived inputs are limited to those specific to a given batch request and contain information used to populate the active templates. Long lived inputs contain configuration policy. These inputs exist in a variety of forms, including the default rules and data models used in both steps 1 and 2. The policy data model captures rarely changing information, such as the number of ports on an line access card, as well as stable configuration parameters, e.g., domain wide network access lists. The policy data model also encompasses the library of templates for configlets and whole configurations. As noted above, the templates describe the comprehensive configuration policy logic in the native device configuration language. Typically, new templates are released after significant scrutiny and test on a release schedule (albeit at a rapid pace), accompanied in parallel with customer or user notification – as template change leads to change in network and service behavior.

All data models eventually require maintenance, including the policy data model. PRESTO simplifies maintaining tabular supplementary data on hardware configuration rules, and PRESTO language templates by storing both in a database. We found that tabulating hardware configuration data only allows for easy additions of new hardware options, but it also allows non-technical domain experts to update tables by maintaining and submitting corresponding spreadsheets. As noted in Section 4, the active templates are also broken down into configlets (sub-templates). The configlet concept allows logical components to be easily updated without affecting the entire library.

5.2 Validating Step-2 Inputs

Step 1 cleans up and normalizes short lived input data. PRESTO provides an interface between steps 1 and 2 to allows users to update and validate values to ensure all required data is available and correct. Users, however, can make mistakes. Therefore, step 2 must perform sanity checks; both syntactic and semantic checks are required. Syntax checks occur in the parsing phase and test for data formats, e.g., an IP addresses are valid "dotted quads," of the form *x.x.x.x/y*. Factoring parsing checks to the parsing phase of step 2 simplifies the system and improves template readability – as the templates are written under the assumption that domain agnostic inputs such as IP addresses are syntactically correct. Semantic checks are more implemented naturally in the configlets within the PRESTO language, as these checks are domain or application specific, e.g., a check may assure that an IP address is neither a network or broadcast address. To support these various forms of checks, the PRESTO language provides capabilities to emit errors and warnings, via functions `templateError()` and `templateWarning()`, each taking a string argument.

5.3 Handling Unknown Information

We found warning messages to be of remarkable utility, as PRESTO users insisted that the process of generation router configuration files proceed in spite of missing information. Users often preferred to obtain configurations with warnings that some information might be missing or inferred, rather than obtaining just an error message. As this eventuality may appear surprising given PRESTO's two step architecture, some explanation may be in order. Routers for a given project or customer are not always ordered or provisioned at the same time. As routers are related to one another through their configuration, situations sometimes arise where information needed to configure one set of routers may depend on the ordering and

configuration of a second set of routers, and the information flow for this second set may be missing at the time that PRESTO is invoked to provision the first set. In other scenarios, essential parameters such as IP addresses which must be written into the configuration may be unknown at the time of initial configuration generation.

PRESTO accommodates these scenarios by allowing the active templates to perform a sort of due diligence. Active templates perform conditional checks to see if all non-mandatory inputs are available. To allow data availability tests of values that drive iteration or looping constructs, the PRESTO template language was extended to support query checking by allowing the standard SQL syntax `count(*) as COUNT`. Using such a query, the new context has access to a `COUNT` variable, on which conditional logic is performed, allowing for the detection of missing information. Where these non-mandatory inputs are missing, configuration lines are still generated with dummy or inferred values, but language specific comments ensure the produced code is still of some utility (e.g., the router will boot and provide basic connectivity), leaving to automation downstream of PRESTO to complete the configuration task.

5.4 Implementation

The core PRESTO system was implemented in approximately 3,000 lines of Perl code. The code is divided into two modules, `PROVGEN.pm` and `PROVDB.pm`, which implement the language interpreter and database interface, respectively. To accommodate a new application, we found the application specific adapters to deal with step 1 inputs can easily grow to thousands of lines. Fortunately, however, many identical components or parsing patterns can be easily ported from one application to another.

6 Experience with Real Services

The development of the PRESTO system has benefited from the insights of network designers and engineers responsible for configuring network elements for large commercial connectivity services. A key measure of the value of such a tool ultimately is how useful and usable it is in practice for this target user community. The PRESTO system is currently being used to automate configuration generation for a number of different commercial network services. In addition to the clear pressing benefits of a successful configuration tool for network management, such an exercise is important for the following reasons:

- It helps us understand the type and amount of effort needed in the process of taking the PRESTO tool and customizing it for a new service.
- It allows us to evaluate and if required evolve the design of the tool in the context of a real world application and its particular requirements.
- It provides valuable feedback for improving the tool.

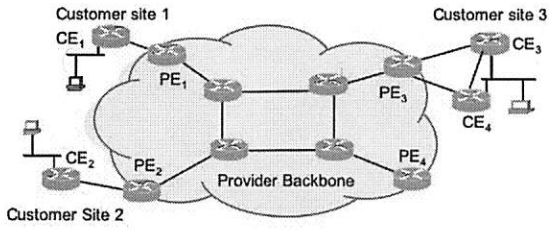


Figure 4: Provider VPN

In this section we present our experiences with, and lessons learned from automating configuration for provider-based VPNs via PRESTO. We first describe the service, then outline our experiences with customizing PRESTO for this service.

6.1 Customer edge router configuration for provider VPNs

Enterprise networks are increasingly using provider-based Virtual Private Networks (VPNs) to connect geographically disparate sites. In such a VPN, each customer site has one or more customer edge (CE) router connecting to a one or more provider edge (PE) routers in the provider network (see Figure 4). Incoming customer traffic from a CE is encapsulated at the PE and carried across the provider network, decapsulated by a remote provider edge router and handed off to the customer CE router at a remote site of the same customer. Traffic belonging to different customers is segregated in the provider backbone, and the provider network is opaque to the customer. The predominant method for supporting provider-based VPNs uses MPLS as the encapsulating technology across the provider backbone.

From a provider perspective, a critical part of supporting the VPN service involves configuring the the CE routers. The tasks include configuring ACLs, interfaces, WAN (Wide Area Network) and LAN (Local Area Network) links and routing (e.g., OSPF and BGP) . The key challenges pertain to heterogeneity and scale. VPN services enjoy a large and growing number of customers. A single customer can have hundreds to thousands of different sites. There are a wide range of features and options a customer can select that impact the configuration.

There are different hardware elements (router models, interface cards), different access type options for the CE-PE connection (e.g., Frame Relay and ATM), different interface speed options, and so on. Customers can opt for a range of resiliency options for each of their sites, where the resiliency option determines the number of routers at the site, the number of distinct links from the site towards the provider network, whether there is a last-resort dial backup to be used if the data service fails, etc. An example resiliency option is 2 CEs with 2 different links to the provider network running in load sharing mode. In addition to being already large, the features and options also change as the service offering improves and evolves. For instance, newer router models and line cards are being constantly added to the list of supported options.

The CE configuration task embodies many of the challenges outlined in Section 2. VPN services involve fast growing demand, a large and increasing volume of router orders to be configured, a huge space of feature combinations, and the need to support a steadily increasing slew of new features. All these factors make the overheads with a manual-intensive configuration workflow unacceptably high, and make these services prime candidates for PRESTO automation.

6.2 A PRESTO tool for CE configuration

Developing a PRESTO-based CE configuration tool involved knowledge engineering (codifying expert knowledge, initially only partially documented) and data modeling (identifying service-specific information and business rules), as well as front end user interface and back end PRESTO template development. The main tasks involved were:

- identifying all the service-specific information required for building the CE configs, and developing the resulting VPN-specific configuration data model.
- understanding the workflow surrounding the provisioning process and available data sources and determining how the information in the above data model can be extracted.
- collating the service-specific provisioning rules and building the service-specific templates based on engineering guidelines from the service designers.
- defining the workflow of the PRESTO-based tool and developing service-specific code around the core service-agnostic PRESTO system.

Recall that PRESTO requires an application to define two types of data models—a provisioning data model for short lived data, and a policy data model for supplemental data and templates. The provisioning data model provides a normalized repository of data specific to the current router request set. The VPN instantiation of the provisioning data model placed as many fields as possible in a central `ROUTER` table indexed by the router hostname. This contained router specific information such as model number, software version, and available customer information. Whenever multiple instances of any type of data was required to build template iterations, a new table was created – that is the data model was highly normalized, as replication has risk in provisioning tasks. For VPN, this led to the creation of tables for WAN interfaces, dial backup information, and the logical interfaces that define VPN connections. In total, the resulting provisioning data model consisted of one main `ROUTER` table and nine secondary tables, each containing foreign keys to reference the `ROUTER` table.

The longer lived policy data model was split into supplemental data and template data sub-models. Supplemental data consisted mainly of data already naturally expressed in tabular form, e.g., mappings from card names to interface type and number of ports, and mappings from strings describing interface speeds to the actual value to code in the configuration. The template data model proved much more interesting, as it contained the configlets used to create the actual router configuration. Configlets were grouped by logical features, specifically `BASE`, `LAN`, `WAN`, `RESILIENCY`, `DAC`, and `B2B`. as follows. The `BASE` table consisted of the configlet required for all routers, e.g., hostname, password, loopback, and `motd` commands. The `LAN` and `WAN` tables contained configlets for types of interfaces, e.g., frame relay and ATM interfaces. The `RESILIENCY` table contained configlets defining the different resiliency options required by the VPN service. The `DAC` table contained configlets specific to various parts of the dial backup configuration. The `B2B` table defined special interface definitions used where CE routers are organized in back to back configurations. Defining these new feature tables as primitives or building blocks allowed specialized configlets to be easily composed and promoted knowledge and code reuse. A total of 44 configlets containing 5414 lines of statements were created.

6.3 Benefits and Experiences

Various existing processes such as accounting, customer service, provisioning, and network management interact with configuration management. For the PRESTO tool to be practically usable, it was critical for it to operate within the confines of the surrounding existing configuration management processes and systems. This requirement to operate in pre-existing existing system and

tool environments significantly impacted the PRESTO design, and the extent to which the configuration generation could be automated. Indeed, the 2-step PRESTO architecture and the accommodations for potential human intervention/oversight between the two Steps were key design elements that resulted from this requirement.

We next revisit the requirements criteria introduced in Section 2.2 and discuss to what extent the PRESTO realization achieves those.

• *Support existing configuration languages:* The PRESTO template language achieved this goal completely. A PRESTO template consists of active template code and configuration statements in an existing configuration language. The template language was used to extract the particular configuration context, determine the control of flow in a configlet, specify rules for combining configlets, and achieve variable substitution and functional substitution. However, the actual configlet was specified in the configuration language that network engineers are familiar with, e.g., Cicso CLI (Figure 2).

• *Scale with options, combinations, and infrastructure:* Several aspects of the PRESTO design made it possible to write a configlet once and reuse it for many different configuration scenarios. These included the capability to dynamically extract data as needed, the approach of writing small configlets for specific features of a configuration, and the support provided for combining configlets together deterministically or conditionally. Reuse of configlets was critical in ensuring that the authoring of templates for the VPN service was tractable, despite the large number of feature combinations in this service. Our experience demonstrated that the effort required to author templates the first time was acceptable – any significant lags were attributable to legitimate debate on the nuances of the precise design intent and associated router capabilities. The incremental effort in updating the system to handle new features was also low. Any updates to the supplementary data such as a new interface card were realized by simply updating the relevant table in the database, without any additional coding effort. For supporting a new router model, we were able to reuse all the existing templates for common features, and only needed to write templates for features that were not yet covered (e.g., a new interface type) or that were router model specific (e.g., rules for numbering interfaces). In fact, the ability to reuse existing templates has proven to be a key strength of the PRESTO approach.

• *Support heterogeneous and diverse data sources:* A key task involved modeling the information needed for service configuration and determining how that information could be obtained from existing data sources. For the VPN service, there were multiple sources of input information: (i) a customer order document that contained details about a customers request for the service, such as a list of sites, and the selection of choices for that site (as listed above, the choices included the number of routers, router models, number of access links, access types, and resiliency option); (ii) other documents that listed required auxiliary information, e.g., a list of the supported router models and cards and the type and number of interfaces on them, (iii) engineering policy documents that specified the configuration rules for all combinations of customer order options. While a large subset of the required information could be directly gleaned from the various data sources described before, there were other important information pieces that for different reasons could not be pulled automatically from an external source. Some of this information required application of service-specific business rules and computations to available input data, while other information needed to be manually assigned. We found that the 2-step PRESTO architecture (see Section 5.1) was well suited to handle this data-heterogeneous environment. In addition to getting available information from a variety of input sources, Step 1 marshalled additional required values and choices in the data model by applying service-specific rules to the available input information. The resulting partially populated provisioning data model was exposed to the user at the end of Step 1 for validation and for filling in missing values. Step 2 of the system then successfully drove the task of actually creating the CE configurations.

One measure of the benefit an automation tool is the reduction in the amount of human mediated effort. While an exact apple to apple comparison is not easy, we compared the traditional VPN CE configuration workflow to the PRESTO workflow. In the traditional manual configuration workflow, engineers proceeded through a manual time consuming process where they collated the different data sources, ran complex computations to derive additional necessary information, and then navigated the complex options in the customer order to create the router configuration. In a customer order with many sites and routers, the process had to be repeated many times, once for each router. If dependencies existed between multiple routers, engineers had also to be careful to reflect the dependencies and build consistent configurations. For instance, if 2 routers had a connection between them, the configuration of the interfaces on both sides should be consistent. In contrast, the PRESTO tool for CE configuration has a significantly more automated workflow, where:

1. The configuration engineer uploaded the customer order, and begins executing the first step of PRESTO.

2. PRESTO created a document at the end of the first step that contained for each router in the customer

order a list of all the information fields needed for configuring the router. Of the total of 64 relevant fields, about 42 were auto-populated with values parsed directly from the various input sources, and another 14 were auto-populated by applying default engineering rules coded into PRESTO. A small number of fields (around eight), had to be manually filled in.

3. The engineer reviewed this document, filled in the missing field values, overrode auto-populated values if required, and initiated the execution of the second step of PRESTO.

4. The tool then created and returned the configurations for all the routers in the customer order.

The manual effort with PRESTO was reduced to filling in a small number of values per router, reviewing auto-populated fields, and sometimes overriding them. Feedback from user trials indicated that engineers found the approach of auto-computing, and auto-populating field values based on default rules to be very useful, even though manual intervention was sometimes needed to override the values.

The PRESTO CE configuration tool was put through user trials involving engineers from across the world. This helped uncover and normalize certain configuration rules that showed regional variations under the manual process, reinforcing the need for the PRESTO tool. One lesson from the user trials was that the system must not only create correct configurations, but also must support a streamlined and sparse user interface. Though we do not describe the details here, designing a suitable user interface proved non-trivial, and required several iterations before it passed the acceptance threshold of users.

7 Related Work

Several industrial products (for example, [6, 16, 20, 21, 7]) have emerged that offer support for configuration management. Many of these efforts have focused on developing abstract languages to specify configurations in a vendor neutral fashion, e.g., IETF standard SNMP5D MIBs [5], and the Common Information Model (CIM) [9]. These information models define and organize configuration semantics for networking/computing equipment and services in a manner not bound to a particular manufacturer or implementation. An example of the success of such an approach is the DSL Forum's TR-069 effort for DSL router configuration [10]. Yet, general router configuration via this approach is challenging given rapid technology evolution, forces driving network operators and vendors towards competitive and differentiated advantage, feature proliferation, and the need to continuously expand networks and features while maintaining backwards compatibility.

Boehm et. al. [3] present a system that raises the abstraction level at which routing policies are specified from individual BGP statements to a network-wide routing policy. The system includes support to generate the appropriate pieces of router configuration for all routers in the network. An approach to automated provisioning of BGP-speaking customers is discussed in [15]. These efforts focus on BGP, just one component of router configuration. Narain [18] seeks to bridge the gap between end-to-end network service requirements, and detailed component configurations, by formal specification of network service requirements. Such specification could aid synthesis of router configurations. In contrast to these efforts, our focus in PRESTO is on the synthesis of complete, precise, and diverse network configurations that are readily deployable.

Several initiatives have explored configuration management systems for desktop, and server environments [1, 4, 2, 12]. Networked and router environments often involve more complex options and interdependencies than desktop environments, and these solutions do not directly apply. That said, there is much potential benefit from cross-fertilization between these domains. Further, many of these works have emphasized deployment of configurations, and have placed relatively little effort on deciding what the configuration of a node should be [2].

While the focus of PRESTO is the synthesis of configuration files, others have looked at important orthogonal issues related to configuration management. The Network Configuration Protocol (NETCONF) [19, 11] effort provides mechanisms to install, manipulate, and delete the configuration of network devices. The NESTOR project [22] seeks to simplify configuration management tasks which requires changes in multiple interdependent elements at different network layers by avoiding inconsistent configuration states among elements, and facilitating undo of configuration changes to recover an operational state. Others [13, 14] have looked at detailed modeling and detection of errors in deployed configurations.

8 Conclusions

The PRESTO system presented throughout represents a step toward realistic automation of massive scale configuration management. While its genesis was mandated by the specific needs of a single provider, the approach and insights are universal. Central to the success of PRESTO are the satisfied mandates for the treatment of complex and evolving service definitions and customer requirements, dealing with the hugely diverse and sometimes

unreliable data sources, and communication within the *lingua franca* of its user community.

PRESTO attempts to balance these requirements by providing malleable and composable *configlets* that encode configuration business logic directly in the target language. Integration of external data sources is performed by simple code embedded in templates and specialized database query interfaces. This approach provides network engineers with rigorous tools to clearly define the workflow and content of service configuration while maintaining the flexibility of enablers to manipulate the resulting configurations to suit the customer or environment in which they will be used. Additionally, configlets are not just applicable for routers; other devices with text-based configuration can also benefit.

Our experiences with PRESTO in the VPN and other services are promising. We learned much that led us to adjust the language and the way it is used in practice, but also confirmed that PRESTO approach is viable. However, we found one of the greatest challenges of automating router configuration at a massive scale is the ability to gather and reconcile necessary input data.

Our future work extends PRESTO in two key directions. First, we are deploying the tool in a wider range of services. Our goal here is to demonstrate that much of the PRESTO design is general, and new services supported with relatively little effort. The second major initiative is to evaluate PRESTO as a platform for "brownfield" configuration in full generality, where updates may be made by a set of systems and tools, with PRESTO among this set. Such tools tackle the more complex problem of projecting a configuration into a live system without negative consequences, e.g., causing performance, security, or connectivity problems. These efforts may require significantly more intelligence than the smart templates currently supported, and may require the introduction of techniques that reason about the consequences of configuration with respect to the services that are already present. It is through these efforts that providers will begin to ease the burden of costly and error-prone configuration management.

Notes

[1]The specification may in fact be split across more than one file, or modality of description of the command set

[2]*Enablers* are the personnel who implement a given service, either staff on-site or within a provider's Network Operations Center.

[3]Note, this use of *context* is different than in context-based evaluation of programming language literature.

References

[1] P. Anderson. Towards a high-level machine configuration system. In *Proc. of the 8th Large Installations Systems Administration (LISA) Conference*, 1994.

[2] P. Anderson and E. Smith. Configuration tools: Working together. In *Proc. of the Large Installations Systems Administration (LISA) Conference*, December 2005.

[3] H. Boehm, A. Feldmann, O. Maennel, C. Reiser, and R. Volk. Network-wide inter-domain routing policies: Design and realization. April 2005.

[4] M. Burgess. Cfengine: a site configuration engine. In *USENIX Computing systems, Vol 8, No. 3*, 1995.

[5] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (snmp). `http://www.ietf.org/rfc/rfc1157.txt`, May 1990.

[6] Cisco IP solution center. `http://www.cisco.com/en/US/products/sw/netmgtsw/ps4748/index.html`.

[7] Cisco Systems Inc. Cisco works small network management solution version 1.5. `http://www.cisco.com/warp/public/cc/pd/wr2k/prodlit/snms_ov.pdf`, 2003.

[8] Cisco Systems, Inc. *Cisco IOS Configuration Fundamentals Command Reference*, 2006. Release 12.4.

[9] Distributed Management Task Force, Inc. `http://www.dmtf.org`.

[10] DSL forum TR-069. `http://www.dslforum.org/aboutdsl/tr_table.html`.

[11] R. Enns. NETCONF configuration protocol. `http://www.ietf.org/internet-drafts/draft-ietf-netconf-prot-12.txt`, February 2006.

[12] N. Desai et al. A case study in configuration management tool deployment. In *Proc. of the Large Installations Systems Administration (LISA) Conference*, December 2005.

[13] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proc. of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[14] A. Feldmann and J. Rexford. IP network configuration for intradomain traffic engineering. In *IEEE Network Magazine*, September 2001.

[15] J. Gottlieb, A. Greenberg, J. Rexford, and Jia Wang. Automated provisioning of BGP customers. In *IEEE Network Magazine*, December 2003.

[16] Intelliden. `http://www.intelliden.com/`.

[17] T. Li, B. Cole, P. Morton, and D. Li. RFC 2281, Cisco hot standby router protocol (HSRP). *Internet Engineering Task Force*, March 1998. `http://www.ietf.org/rfc/rfc2281.txt`.

[18] S. Narain. Network configuration management via model finding. In *Proc. of the Large Installations Systems Administration (LISA) Conference*, December 2005.

[19] Network configuration (netconf). `http://www.ietf.org/html.charters/netconf-charter.html`.

[20] Opsware. `http://www.opsware.com/`.

[21] Voyence. `http://www.voyence.com/`.

[22] Yechiam Yemini, Alexander Konstantinou, and Danilo Florissi. NESTOR: An architecture for network self-management and organization. *IEEE Journal on Selected Areas in Communications*, 18(5):758–766, May 2000.

Events Can Make Sense

Maxwell Krohn (*MIT CSAIL*), Eddie Kohler (*UCLA*) and M. Frans Kaashoek (*MIT CSAIL*)
{krohn,kaashoek}@csail.mit.edu, kohler@cs.ucla.edu

ABSTRACT

Tame is a new event-based system for managing concurrency in network applications. Code written with Tame abstractions does not suffer from the "stack-ripping" problem associated with other event libraries. Like threaded code, tamed code uses standard control flow, automatically-managed local variables, and modular interfaces between callers and callees. Tame's implementation consists of C++ libraries and a source-to-source translator; no platform-specific support or compiler modifications are required, and Tame induces little runtime overhead. Experience with Tame in real-world systems, including a popular commercial Web site, suggests it is easy to adopt and deploy.

1 INTRODUCTION

This paper introduces Tame, a system for managing concurrency in network applications that combines the flexibility and performance of events with the programmability of threads. Tame is yet another design point in a crowded space, but one that has proven successful in real-world deployments. The system is, at heart, an event-based programming library that frees event developers from the annoyance of "stack ripping" [1]. We have implemented Tame in C++ using libraries and source-to-source translation, making Tame deployable without compiler upgrades.

Threads are the more popular strategy for managing concurrency, but some situations (and programmers) still call for events [7, 13, 24, 28, 34, 39]. Applications with exotic concurrency, such as multicast, publish/subscribe, or TCP-like state machines, might find threads insufficiently expressive [37]. Certain contexts do not support threads or blocking [6, 21]. On new platforms, portability can favor events, which require only a `select` call and no knowledge of hardware-specific stack or register configuration [11]. Finally, some event-based servers perform better and use less memory than threaded competitors [18, 24–26].

But a key advantage of events—a single stack—is also a liability. Sharing one stack for multiple tasks requires stack ripping, which plagues the development, maintenance, debugging and profiling of event code [1]. The programmer must manually split (or "rip") each function that might block (due to network communication or disk I/O), as well as all of its ancestors in the call stack. Ripping a function obscures its control flow [6] and complicates memory management.

However, the right abstractions can capture events' expressivity while minimizing the headaches of stack ripping [30]. The Tame system introduces powerful abstractions with implementation techniques suitable for high-performance system programming. The specific contributions of the Tame system are:

1. A high-level, type-safe API for event-based programming that frees it from the stack-ripping problem but is still backwards compatible with legacy event code.
2. A new technique to incorporate threads and events in the same program.
3. A maintainable and immediately deployable implementation in C++, using only portable libraries and source-to-source translation.
4. An automated memory management scheme for events that does not require garbage collection.

Our experience with Tame has shown the interface sufficient to build and run real systems. Programmers other than the authors rely on Tame in educational assignments, research projects [36], and even a high-traffic commercial Web site [16].

2 RELATED WORK

The research systems most closely related to Tame are Capriccio [38] and the work of Adya et al. [1]. Capriccio is a cooperative threading package that exports the POSIX thread interface but looks like events to the operating system: it uses sophisticated stack management to make one stack appear as many, saving on cycles and memory. However, the Capriccio system strives to equal events only in terms of *performance* and not in terms of *expressivity*; its authors note that the thread interface is less flexible than that of events [37].

Adya et al.'s system is a way to combine event-based and threaded code in the same address space. The key insight is that a program's style of stack management (automatic or manual) is orthogonal to its style of task management (cooperative or preemptive) and that most literature on events and threads mistakenly claims they are linked. As in Adya's system, a Tame program can be expressed in a syntax that has readable automatic stack management (like threads) yet has explicit cooperative task management (like events). Tame differs because it extends automatic stack management to *all* event code, while "hybrid" event code in Adya's system still requires manual stack management. Other systems like SEDA [40] use threads

and events in concert to achieve flexible scheduling and intraprocess concurrency. Tame is complementary to such hybrid systems and can be used as an implementation technique to simplify their event code.

Many other systems attempt to improve threads' scalability and efficiency. NPTL in Linux [9] and I/O completion ports in Windows [22] improve the performance of kernel threads; we compare Tame with NPTL in our evaluation. Practical user-level cooperative threading packages include Gnu PTH, which focuses on portability [12], and StateThreads, which focuses on performance [31].

Existing practical event libraries fall into several categories. The most primitive, such as `libevent` [27], focus exclusively on abstracting the interface to OS events (i.e., `select` vs. `poll` vs. `epoll` vs. `kqueue`), and don't simplify the construction of higher-level events, such as RPC completions. The event libraries integrated with GUI toolkits, such as Motif, GTK+, and Qt, support higher-level events, but are of course tuned for GUIs rather than general systems programming. The type-safe *libasync* event library for C++ is the basis of our work [21, 41].

The protothreads C-preprocessor library [11] gives the illusion of threads with only one stack. Protothreads are useful in resource-constrained settings such as embedded devices and sensor networks, but lacking stacks or closures, they must use global variables to retain state and therefore are not suited to building composable APIs. The Tame system shares implementation techniques with protothreads and similar C coroutine libraries [10, 11], as well as the `porch` program checkpointer [29].

The Tame language semantics draw from a rich body of previous work on parallel programming [32]. Like condition variables [14], Tame's `events` allow signaling and synchronization between different parts of a program, but unlike condition variables, `events` do not require locks (or threads, for that matter). Many parallel programming languages have constructs similar to Tame's `twait`: Occam has `PAR` [17], and Pascal-FC has `COBEGIN` and `COEND` [8].

Tame also borrows ideas such as closures and function currying from functional languages like Lisp [33] and Haskell [15]. Previous work in modeling threads and concurrency in functional languages, such as Haskell and ML, has noted a correspondence between continuations and threads. A user-level thread scheduler essentially chooses among a set of active continuations; blocking adds the current continuation to this set and invokes the scheduler. For instance, Claessen uses monads in Haskell to implement threading [5]. Li and Zdancewic extend Claessen's technique to combine threads and events [20]. Concurrent ML (CML) uses continuations to build a set of concurrency primitives much like those of Tame [30]. Tame and CML have similar `events`, Tame's `rendezvous` shares some properties with CML's `choose` operator, and Tame's `twait` is analogous to CML's `sync`. There are differences in performance and function. CML `events` are effectively continuations and preserve the equivalent of an entire call stack, while Tame `events` preserve only the top-level function's closure, and CML has no direct equivalent for Tame's user-supplied event IDs—instead the CML user must manipulate `event` objects directly. Tame's constructs have similar power but are efficiently implementable in conventional systems programming languages like C++.

```
// Threads
void wait_then_print_threads() {
   sleep(10);          // blocks this function and all callers
   printf("Done!");
}

// Tame primitives
tamed wait_then_print_tame() {
    tvars { rendezvous<> r; }
    event<> e = mkevent(r);       // allocate event on r
    timer(10, e);    // cause e to be triggered after 10 sec
    twait(r);        // block until an event on r is triggered
                  // only blocks this function, not its callers!
    printf("Done!");
}

// Tame syntactic sugar
tamed wait_then_print_simple_tame() {
   twait { timer(10, mkevent()); }
   printf("Done!");
}
```

Figure 1: Three functions that print `Done!` after ten seconds. The first version uses threads; the second Tame version is essentially as readable.

3 Tame Semantics

Tame makes easy concurrency problems easy to express in events (as they were easy to express in threads). Figure 1 shows three implementations of a trivial function; the second Tame version is indeed close to the threaded version in code length and readability. The rest of this section describes the Tame primitives and syntactic sugar. We also show through examples how the full power of Tame simplifies the expression of *hard* concurrency problems, and how Tame allows users to develop composable solutions for concurrency problems (harder to express correctly in threads).

3.1 Overview

Tame introduces four related abstractions for handling concurrency: *events*, *wait points*, *rendezvous*, and *safe local variables*. They are expressed as software libraries whenever possible, and as language extensions (via source-to-source translation) when not.

First, each **event** object represents a future occurrence, such as the completion of a network read. When the expected occurrence actually happens—for instance, a packet arrives—the programmer *triggers* the event by calling its `trigger` method.

The `mkevent` function allocates an event of type `event<`*T*`>`, where *T* is a sequence of zero or more types.

This event's `trigger` method has the signature `void trigger(T)`. Calling `trigger(v)` marks the event as having occurred, and passes zero or more results *v*, which are called *trigger values*, to whomever is expecting the event. For example:

```
rendezvous<> r;  int i = 0;
event<int> e = mkevent(r, i);
e.trigger(100);
assert(i == 100);                    // assertion will succeed
```

When triggered, e's `int` trigger value is stored in `i`, whose type is echoed in e's type.

The **wait point** language extension, written `twait`, blocks the calling function until one or more events are triggered. Blocking causes a function to return to its caller, but the function does not complete: its execution point and local variables are preserved in memory. When an expected event occurs, the function "unblocks" and resumes processing at the wait point. By that time, of course, the function's original caller may have returned. Any function containing a wait point is marked with the `tamed` keyword, which informs the caller that the function can block.

The first, and more common, form of wait point is written "`twait { `*statements*`; }`". This executes the *statements*, then blocks until *all* events created by `mkevent` calls in the *statements* have triggered. For example, code like "`twait { timer(10, mkevent()); }`" should be read as "execute '`timer(10, mkevent())`', then block until the created event has triggered"—or, since `timer` triggers its event argument after the given number of seconds has passed, simply as "block for 10 seconds". `twait{}` can implement many forms of event-driven control flow, including serial and parallel RPCs.

The second, more flexible form of wait point explicitly names a **rendezvous** object, which specifies the set of expected events relevant to the wait point. Every `event` object associates with one `rendezvous`. A wait point `twait(r)` unblocks when *any one* of `rendezvous r`'s events occurs. Unblocking consumes the event and restarts the blocked function. The first form of wait point is actually syntactic sugar for the second: code like "`twait { `*statements*`; }`" expands into something like

```
{ rendezvous<> __r;
  statements;  // where mkevent calls create events on __r
  while (not all __r events have completed)
    twait(__r);                                    }
```

The `twait()` form can also return information about *which* event occurred. A rendezvous of type `rendezvous<`*I*`>` accepts events with *event IDs* of type(s) *I*. Event IDs identify events in the same way thread IDs identify threads, except that event IDs have arbitrary, programmer-chosen types and values. A `twait(r, `*i*`)` statement then sets *i* to the ID(s) of the unblocking event.

Although wait points are analogous to blocking a thread until a condition variable is notified, blocking in Tame has a different meaning than in threads. A blocked threaded function's caller only resumes when the callee explicitly `returns`. In Tame, by contrast, a `tamed` function's caller resumes when the called function *either returns or blocks*. To allow its caller to distinguish returning from blocking, a `tamed` function will often accept an event argument, which it triggers when it returns. This trigger signals the function's completion. Here is a function that blocks, then returns an integer, in threads and in Tame:

```
int blockf() {
  ... block ...
  return 200;
}

i = blockf();
```

```
tamed blockf(event<int> done) {
  ... block ...
  done.trigger(200);
}

twait { blockf(mkevent(i)); }
```

In Tame, the caller uses `twait` to wait for `blockf` to return, and so must become `tamed` itself. Waiting for events thus trickles up the call stack until a caller doesn't care whether its callee returns or blocks. This property is related to stack ripping, but much simpler, since functions do not split into pieces. Threaded code avoids any such change at the cost of blocking *the entire call stack* whenever a function blocks. Single-function blocking gives Tame its event flavor, increases its flexibility, and reduces its overhead (only the relevant parts of the call stack are saved). We return to this topic in the next section.

When an event `e` is triggered, Tame queues a *trigger notification* for e's event ID on e's rendezvous `r`. This step also unblocks any function blocked on `twait(r)`. Conversely, `twait(r)` checks for any queued trigger notifications on `r`. If one exists, it is dequeued and returned. Otherwise, the function blocks at that wait point; it will unblock and recheck the rendezvous once someone triggers a corresponding event. The top-level event loop cycles through unblocked functions, calling them in round-robin order when unblocking on file descriptor I/O and first-come-first-served order otherwise. More sophisticated queuing and scheduling techniques [40] are possible.

Multiple functions cannot simultaneously block on the same rendezvous. In practice, this restriction isn't significant since most rendezvous are local to a single function. A Tame program that needs two functions to wait on the same condition uses two separate events, triggering both when the condition occurs. Tame-based read locks (see Section 7.5) are an example of such a pattern.

Finally, **safe local variables**, a language extension, are variables whose values are preserved across wait points. The programmer marks local variables as safe by enclosing them in a `tvars {}` block, which preserves their values in a heap-allocated closure. (Function parameters are always safe.) Unsafe local variables have indeterminate values after a wait point. The C++ compiler's uninitialized-

Classes	Keywords & Language Extensions	Functions & Methods

`event<>`
• A basic event.
`event<T>`
• An event with a single *trigger value* of type T. This value is set when the event occurs; an example might be a character read from a file descriptor. Events may also have multiple trigger values of types $T_1 \ldots T_n$.
`rendezvous<I>`
• Represents a set of outstanding events with event IDs of type I. Callers name a rendezvous when they block, and unblock on the triggering of any associated event.
	`twait(r[,i]);`
	• A wait point. Block on explicit rendezvous r, and optionally set the event ID i when control resumes.
	`tamed`
	• A return type for functions that use `twait`.
	`tvars { ... }`
	• Marks safe local variables.
	`twait { statements; }`
	• Wait point syntactic sugar: block on an implicit rendezvous until all events created in *statements* have triggered.
		`mkevent(r,i,s);`
		• Allocate a new event with event ID i. When triggered, it will awake rendezvous r and store trigger value in slot s.
		`mkevent(s);`
		• Allocate a new event for an implicit `twait{}` rendezvous. When triggered, store trigger value in slot s.
		`e.trigger(v);`
		• Trigger event e, with trigger value v.
		`timer(to,e); wait_on_fd(fd,rw,e);`
		• Primitive event interface for timeouts and file descriptor events, respectively.

Figure 2: Tame primitives for event programming in C++.

variable warnings tell a Tame programmer when a local variable should be made safe.

Type signatures Events reflect the types of their trigger values, and rendezvous reflect the types of their event IDs. The compiler catches type mismatches and reports them as errors. Concretely, `rendezvous` is a conventional C++ template type, defined in a library. All events associated with a rendezvous of type `rendezvous<I>` must have event IDs of type `I`. The `mkevent` function has type:

```
event<T1,T2,...> mkevent(rendezvous<I> r, const I &i,
    T1 &s1, T2 &s2, ...);
```

The arguments are a `rendezvous`, an event ID `i`, and slot references `s1`, `s2`, ... that will store trigger values when the event is later triggered. C++'s template machinery deduces the appropriate event ID and slot type(s) from the arguments, so `mkevent` can unambiguously accommodate optional event IDs and arbitrary trigger slot types. The `event::trigger` method has type:

```
void event<T1,T2,...>::trigger(const T1 &v1,
    const T2 &v2, ...);
```

When called, this method assigns the trigger values `v1`, `v2`, ... to the slots given at allocation time, then unblocks the corresponding rendezvous. Wait points have type `twait(rendezvous<I> r, I &i);` when the wait point unblocks, `i` holds the ID of the unblocking event.

Primitive events Three library functions provide an interface to low-level operating system events: `timer()`, `wait_on_fd()`, and `wait_on_signal()`. Each function takes an `event<>` e and one or more extra parameters. `timer(`to`,`e`)` triggers e after to seconds have elapsed; `wait_on_fd(`fd`,`rw`,`e`)` triggers e once the file descriptor fd becomes readable or writable (depending on rw); and `wait_on_signal(`sig`,`e`)` triggers e when signal sig is received. The base event loop that understands these functions is implemented in terms of `select()` or platform-specific alternatives such as Linux's epoll or FreeBSD's kqueue [19].

Like all programs based on events or cooperative threads, a tamed program will block entirely if any portion of it calls a blocking system call (such as `open`) or takes a page fault. Tame inherits *libasync*'s non-blocking substitutes for blocking calls in the standard library (such as `open` and `gethostbyname`). For tamed programs to perform well in concurrent settings, they should use only non-blocking calls and should not induce swapping.

Figure 2 summarizes Tame's primitive semantics.

3.2 Control Flow Examples

Common network flow patterns like sequential calls, parallel calls, and windowed calls [37] are difficult to express in standard event libraries but much simplified with Tame. As a running example, consider a function that resolves addresses for a set of DNS host names. An initial design might use the normal blocking resolver:

```
1 void multidns(dnsname name[], ipaddr a[], int n) {
2     for (int i = 0; i < n; i++)
3         a[i] = gethostbyname(name[i]);
4 }
```

Of course, this function will block all other computation until all lookups complete. An efficient server would allow other progress during the lookup process. The event-based solution would use a *nonblocking* resolver, with a signature such as:

```
tamed gethost_ev(dnsname name, event<ipaddr> e);
```

This resolver uses nonblocking I/O when contacting local and/or remote DNS servers. (Alternately, Tame's threading support makes it easy to adapt a blocking resolver for non-blocking use; see Section 4.) Since `gethost_ev`'s caller

```
void multidns_nasty(dnsname name[], ipaddr a[], int n,
                    event<> done) {
   if (n > 0) {
      // When lookup succeeds, gethost_ev will call
      // "helper(name, a, n, done, RESULT)"
      gethost_ev(name[0], wrap(helper, name, a, n, done));
   } else // done, alert caller
      done.trigger();
}
void helper(dnsname *name, ipaddr *a, int n,
            event<> done, ipaddr result) {
   *a = result;
   multidns_nasty(name+1, a+1, n-1, done);
}
```

Figure 3: Stack-ripped *libasync* code for looking up n DNS names without blocking. A simple `for` loop has expanded into two interacting functions, obscuring control flow; all callers must likewise split.

can regain control before the lookup completes, the lookup result is returned via a trigger value: once the address `a` is known, the resolver calls `e.trigger(a)`. The trigger simultaneously exports the result and unblocks anyone waiting for it. Here's how to look up a single name with `gethost_ev`:

```
tvars { ipaddr a; }
twait { gethost_ev(name, mkevent(a)); }
print_addr(a);
```

Without Tame, adapting `multidns` to use `gethost_ev` is an exercise in stack-ripping frustration; for the gory details, see Figure 3. Tame, however, makes it simple:

```
1  tamed multidns_tame(dnsname name[], ipaddr a[],
                       int n, event<> done) {
2     tvars { int i; }
3     for (i = 0; i < n; i++)
4        twait { gethost_ev(name[i], mkevent(a[i])); }
5     done.trigger();
6  }
```

`multidns_tame` keeps all arguments and the local variable `i` in a closure. Whenever `gethost_ev` looks up a name, it triggers the event allocated on line 4. This stores the address in `a[i]` and unblocks `multidns_tame`, after which the loop continues. Though the code somewhat resembles threaded code, the semantics are still event-driven: `multidns_tame` can return control to its caller before it completes. Thus, it signals completion via an event, namely `done`. Any callers that depend on completion must use Tame primitives to block on this event, and thus become `tamed` themselves. The `tamed` return type then bubbles up the call stack, providing the valuable annotation that `multidns_tame` and its callers may suspend computation before completion.

`multidns_tame` allows a server to use the CPU more effectively than `multidns`, since other server computation can take place as `multidns_tame` completes. However, `multidns_tame`'s lookups still happen in series: lookup i does not begin until lookup $i-1$ has completed. The obvious latency improvement is to perform lookups in parallel. The tamed code barely changes:

```
1  tamed multidns_par(dnsname name[], ipaddr a[],
                      int n, event<> done) {
2     twait {
3        for (int i = 0; i < sz; i++)
4           gethost_ev(name[i], mkevent(a[i]));
5     }
6     done.trigger();
7  }
```

The only difference between the serial and parallel versions is the ordering of the `for` and `twait` statements (and that `i` doesn't need to be in the closure). Since both versions have the same signature, the programmer can switch implementation strategies without changing caller code. With threads, however, the serial version could use one thread to do all lookups, while the parallel version would use as many threads as lookups. Tame preserves events' flexibility while providing much of threads' readability.

A generalization of serial and parallel control flow is *windowed* or *pipelined* control flow, in which n calls are made in total, and at most $w \leq n$ of them are outstanding at any time. For serial flow, $w = 1$; for parallel, $w = n$. Intermediate values of w combine the advantages of serial and parallel execution, allowing some overlapping without blasting the server. With Tame, even windowed control flow is readable, although the simplified `twait{}` statement no longer suffices:

```
1  tamed multidns_win(dnsname name[], ipaddr a[],
                      int n, event<> done) {
2     tvars { int sent(0), recv(0); rendezvous<> r; }
3     while (recv < n)
4        if (sent < n && sent - recv < WINDOWSIZE) {
5           gethost_ev(name[sent], mkevent(r,a[sent]));
6           sent++;
7        } else {
8           twait(r);
9           recv++;
10       }
11    done.trigger();
12 }
```

The loop runs until all requests have received responses (`recv == n`). On each iteration, the function sends a new request (lines 5–6) whenever a request remains (`sent < n`) and the window has room (`sent - recv < WINDOWSIZE`). Otherwise, the function harvests an outstanding request (lines 8–9). Again, the signature is unchanged, and the implementation is short and clear. We have not previously seen efficient windowed control flow expressed this simply.

3.3 Typing and Composability

Tame's first-class events and rendezvous, and its distinction between event IDs and trigger values, improve its flexibility, composability, and safety.

First, Tame preserves safe static typing without compromising flexibility by distinguishing event IDs from trigger values. Event IDs are like names. They identify events,

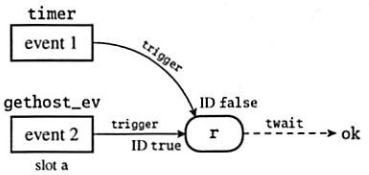


Figure 4: Relationships between events (boxes) and rendezvous (round box) for a DNS lookup with timeout.

and are known when the event is registered; all events on the same rendezvous must have the same event ID type. Trigger values, on the other hand, are like results: they are not known until the event actually triggers. Examples include characters read from a file descriptor, RPC replies, and so forth. Event IDs and trigger values are related, of course; when a `twait` statement returns event ID *i*, the programmer knows that event *i* has triggered, and therefore its associated trigger values have been set. In contrast, several other systems return trigger values as part of an event object; the `twait` equivalent returns the object, and extracting its values requires a dynamic cast. The Tame design avoids error-prone casts while still letting a single rendezvous handle events with entirely different trigger value types.

To demonstrate Tame's composability, we'll add timeouts to the following event-based DNS lookup:

```
tvars { ipaddr a; }
twait { gethost_ev(name, mkevent(a)); }
```

We want to cancel a lookup and report an error if a name fails to resolve in ten seconds. The basic implementation strategy is to wait on *two* events, the lookup and a ten-second timer, and check which event happens first.

```
tvars { ipaddr a;  rendezvous<bool> r;  bool ok; }
timer(10, mkevent(r, false));
gethost_ev(name, mkevent(r, true, a));
twait(r, ok);
if (!ok) printf("Timeout");
r.cancel();
```

The event ID `false` represents timeouts, while `true` represents successful lookup. The `twait` statement sets `ok` to the ID of the event that triggers first[1], so `ok` is false if and only if the lookup timed out. The `r.cancel()` call cleans up state associated with the event that did not trigger. Figure 4 diagrams the relevant objects.

This code is verbose and hard to follow. Supporting timeouts on *every* lookup, or on other types of event, would require adding `rendezvous` and `timer` calls across the program and abandoning the `twait{}` syntactic sugar.

[1] In other libraries we have examined, such as CML, the `twait` function would return an opaque, system-chosen ID that the programmer would compare with return values from `mkevent`. Though this works, event IDs are far more convenient, particularly when many events are outstanding.

```
1  template <typename T> tamed
   __add_timeout(event<T> &e_base, event<bool, T> e) {
2     tvars { rendezvous<bool> r; T result; bool rok; }
3     timer(TIMEOUT, mkevent(r, false));
4     e_base = mkevent(r, true, result);
5     twait(r, rok);
6     e.trigger(rok, result);
7     r.cancel();
8  }

9  template <typename T> event<T> add_timeout(event<bool, T> e) {
10    event<T> e_base;
11    __add_timeout(e_base, e);
12    return e_base;
13 }
```

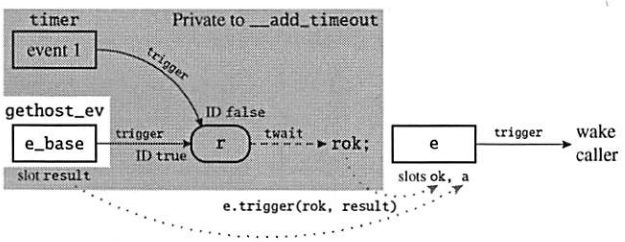


Figure 5: Code and object relationships for a composable timeout event adapter, and its use in a DNS lookup.

Tame can do better. Its programmers can write an adapter that can add a cancellation timeout to *any* event. The adapter relies on C++'s template support and on Tame's first-class events, and resembles adapters from higher-level thread packages such as CML. Many other event libraries could not express this kind of composable abstraction, which was a main motivator for Tame's design. The adapter simplifies the lookup code to:

```
tvars { ipaddr a;  bool ok; }
twait { gethost_ev(name, add_timeout(mkevent(ok, a))); }
if (!ok) printf("Timeout");
```

The caller generates an event with *two* trigger slots, one for the base trigger value and one for a boolean that indicates success or failure. Either a successful lookup or a timeout will trigger the event. Success will set the boolean trigger value to true, timeout will set it to false. Thus, after waiting for the event, callers can examine the boolean to check for timeout. This pattern works with `twait{}` statements as well as explicit rendezvous.

Unfortunately, the `gethost_ev` function requires an event that takes a *single* trigger value, namely the IP address. It will not supply an extra trigger value unless we change its signature and implementation, which would make it specific to our timeout adapter—something we'd like to avoid. But Tame's abstractions let us *transparently interpose* between `gethost_ev` and its "caller". The adapter will set the extra trigger value.

Figure 5 shows the code and a diagram of the object relationships. The real work takes place in `__add_timeout`, which creates *two* events: `e_base`, which is returned (and eventually passed to `gethost_ev`), and an internal event passed to the timer function on line 6. The two

created events associate with the rendezvous `r` local to `__add_timeout`. This is the interposition. When the timeout triggers, or when `e_base` triggers (due to a successful DNS lookup), `__add_timeout` will unblock, set the `ok` slot appropriately, and then trigger `e`. Only this last step unblocks the caller. The caller observes that `ok` and `a` have been set, but is oblivious to `__add_timeout`'s intercession; it is as if `gethost_ev` set `ok` itself.

It would be trivial to add other types of "timeout", such as signal receipt, to `add_timeout`; its signature would not change, and neither would its callers. Similarly, one-line changes could globally track how many events time out. We've added significant additional concurrency semantics with only local changes: the definition of composability.

3.4 Future Work

Tamed processes do not currently run on more than one core or CPU. The production Tame-based applications we know of consist of multiple concurrent processes cooperating to achieve an application goal. OkCupid.com, for instance, uses exclusively multicore and SMP machines. Its Web front-ends run no fewer than fifty site-specific Tame-based processes, all of which simultaneously answer Web requests. When traffic is high, all CPUs (or cores) are in use. Nevertheless, few changes to Tame would be required for true simultaneous threading support. Tame already supports event-based locks to product data structures from unwanted interactions (Section 7.5). As in *async-mp* [41], multiple kernel threads could draw from a shared pool of ready tasks, as restricted by Tame's current atomicity assumption: at most one thread of control can be active in any given closure at a time. Locks enforced by the kernel, or any equivalent technique, could ensure this invariant.

Tame does not currently interact well with C++ exceptions: an exception raised in a Tamed function might be caught by the event loop.

Some of Tame's limitations are not implementation-dependent but rather consequences of its approach and semantics. As mentioned in Section 3, changing a function from a regular C++ function to a tamed function involves signature changes all the way up the call stack. Some developers might object to this limitation, especially those who export libraries with fixed interfaces.

4 Threads

Tame can interoperate with threads when a thread package is available, suggesting that the Tame abstractions (wait points, `events` and `rendezvous`) apply to both programming models. With thread support, Tame simplifies the transition between threaded and event-style programming, for instance allowing event-based applications to use threaded software in the C library (e.g. `gethostbyname`) and database client libraries (e.g. `libmysqlclient` [23]). We have only experimented with cooperative user-level threading packages, though kernel-level threads that support SMPs are also compatible with our approach.

The key semantic difference between threaded and event-based operation is how functions return. In event-based Tame, functions can either return a useful value via a `return` statement or block via `twait`, but not both. Threaded functions *can* both block and return a value, since the caller regains control only when the computation is done.

With thread support, Tame exposes both event and thread return semantics. In Tame, a threaded function is one that calls `twait` but does not have a `tamed` return type. When such a function encounters `twait(r)`, it checks for queued triggers in `r` as usual; if none are present, it asks for a wakeup notification when a trigger arrives in `r`, and then *yields* to another thread. During the yield, the threading package preserves the function's entire call stack (including all of its callers), while running other, more ready computations. When the trigger arrives, the blocked thread awakes at the `twait` call and can return to its caller.

A trivial example using threads in Tame is a reimplementation of the `sleep` call:

```
1  int mysleep(int d) {
2      twait { timer(d, mkevent()); }
3      return d;
4  }
```

As usual, the call to `timer` registers an event that will be triggered after a `d` second delay. The function then calls `twait` on an implicit `rendezvous` at line 2, yielding its thread. After `d` seconds have elapsed, the main thread triggers the `event` allocated on line 2, waking up `mysleep` and advancing control to the `return` statement on line 3. Since `mysleep` is threaded (i.e., does not have a `tamed` return value), it returns an actual value to its caller.

Blocking the current thread uses Tame's existing `twait` syntax, but starting a new thread requires a new `tfork` function:[2]

```
tfork(rendezvous<I> r, I i, threadfunc<V> f, V &v);
```

The semantics are:

1. Allocate `e = mkevent(r, i)`.
2. Fork a new thread. In the new thread context:
   (a) Call `f()` and store its return value in `v`.
   (b) Trigger event `e`.
   (c) Exit the thread.

When the function `f` completes, the `rendezvous` `r` receives a trigger with event ID `i`. This unifies the usually separate concepts of event "blocking" and joining on a

[2] `threadfunc<R>` is an `event` whose trigger method yields a return value of type `R`. Given the function `int f()`, we can create a `threadfunc<int>` from a function pointer to `f`. From the function `int g(int a)`, we can create a `threadfunc<int>` by *wrapping* `g` with an integer argument, as in function currying [15].

thread. Code like the following uses `tfork` and `twait{}` syntactic sugar to call a blocking library function from an event-based context:

```
tamed gethost_ev(const char *name,
                 event<struct hostent *> e) {
  tvars { struct hostent *h; }
  twait { tfork(wrap(gethostbyname, name), h); }
  e.trigger(h);
}
```

This starts `gethostbyname(name)` in a new thread, then blocks in the usual event-driven way until that thread exits. At that point, the caller is notified via an event trigger of the `struct hostent` result.

5 Memory Management

Tame hides most details of event memory management from programmers, protecting them at all costs from wild writes and catching most memory leaks. For the large majority of Tame code that uses the `twait{}` environment, correct program syntax guarantees correct leakless memory management. For more advanced programs that use explicit `rendezvous`, Tame uses reference counting to enforce key invariants at runtime. The invariants are:

I1 A function's closure lives at least until control exits the function for the last time.

I2 Some of an `event`'s trigger slots may be safe local variables, and triggering it assigns values to those variables. Thus, a function's closure must live as least until `events` created in the function have triggered.

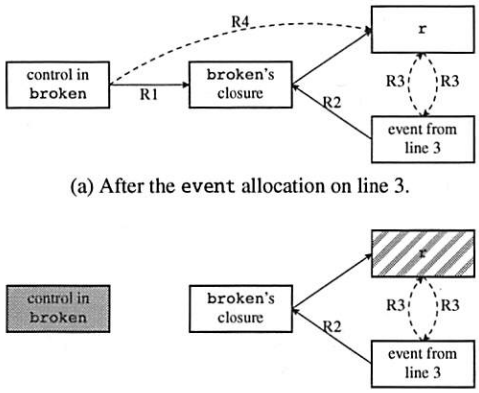
I3 `Events` associated with a `rendezvous` *r* must trigger exactly once before *r* is deallocated. The programmer must uphold I3 by correctly managing `rendezvous` lifetime and triggering each `event` exactly once.

A closure should be deallocated as soon as so doing does not violate I1 or I2.

Of these invariants, I3 depends the most on program correctness. Some cases are easy to handle. Tame ignores attempts to trigger an event multiple times (or aborts, depending on runtime options), and forgetting to trigger an event in a `twait{}` environment will cause a program hang and is thus easily observable. The difficult case involves managing the lifetimes of explicit `rendezvous`. Consider the function `broken`:

```
1  tamed broken(dnsname nm) {
2    tvars { rendezvous<> r;  ipaddr res; }
3    gethost_ev(nm, mkevent(r, res));
4    // Whoops: forgot to twait(r)!
5  }
```

The `event` created on line 3 uses the trigger slot `res`, a safe variable in `broken`'s closure. The function then exits without waiting for `r` or examining `res`. This is a bug—an *event leak* in violation of I3. If Tame deallocated `broken`'s closure eagerly, right after it exited, then the `event`'s eventual trigger would write its value into the deallocated memory where the closure used to be.



(a) After the `event` allocation on line 3.

(b) After control exits `broken` on line 5.

Figure 6: Memory references for the `broken` function. Weak references are shown as dotted lines, and strong references as solid lines. Solid fill indicates a function exit, and striped fill indicates cancellation.

Tame's answer is a careful reference-counting scheme; its runtime keeps track of events and closures with C++ "smart pointer" classes. For example, `event<>` objects are actually smart pointers; the `event` is stored elsewhere, in a private object only accessible by Tame code. If necessary, Tame keeps the `event` around even after the user frees it. There can be circular references among these three types of objects—for example, a closure contains a local `event`, which names a different closure-local variable as a trigger slot. Tame uses two different types of reference counts to break the circularity: *strong* references, which are conventional reference counts, and *weak* references, which allow access to the object only if it hasn't been deallocated.

In outline, Tame keeps the following reference counts:

R1 Entering a tamed function for the first time adds a strong reference to the corresponding closure, which is removed only when the function exits for the last time. This preserves I1.

R2 Each `event` created inside a closure holds a strong reference to that closure, preserving I2. The reference is dropped once the `event` is triggered.

R3 A `rendezvous` and its associated `events` keep weak references to each other. The references a `rendezvous` keeps to its `events` allow it to cancel `events` that did not trigger before the `rendezvous`'s deallocation. Canceling an `event` clears its R2 reference; any future trigger attempt on the `event` will be ignored, preserving I3. The weak references the other way enable an `event` to update its `rendezvous` upon a trigger.

Figure 6a shows these references in the `broken` function following line 3's `event` allocation. The most important problem introduced by this reference counting scheme is due to R2: an untriggered `event` can cause a closure leak.

Such a leak can be caught by checking the associated rendezvous upon deallocation for untriggered events. A rendezvous' deallocation is up to the programmer, but there is an important and common case in which Tame can intervene. If a `rendezvous` was declared as a local variable in some closure, and that closure has exited for the last time, then no future code will call `twait` on the `rendezvous`, *even if the closure cannot be deallocated yet* because of some stray reference. Thus, Tame amends the reference counting protocol as follows:

R4 Exiting a tamed function for the last time cancels any `rendezvous` directly allocated in that function's closure. Canceling a `rendezvous` cancels all `events` associated with it. Actual deallocation occurs only when the closure is deallocated, which might be some time later.

Figure 6b shows how Tame's reference counting protocol solves `broken`'s leak. Control exits the function immediately, forcing `r`'s cancellation by R4. Upon cancellation, `r` checks that all of its `events` have triggered. In this case, the `event` allocated on line 3 has not triggered, but Tame cancels it, clearing its R3, which releases the closure, and in turn, releases `r`. Any eventual trigger of the `event` is ignored.

6 Implementation Details

Tame is implemented as a C++ preprocessor (or source-to-source translator). The difficulty of parsing C++ is well known [2]. Tame avoids as much C++ parsing as possible at the cost of several semantic warts, which could be avoided with fuller compiler integration.

6.1 Closures

Each `tamed` function has one closure with a flat namespace, restricting C/C++'s scoping. Internally, the Tame translator writes a new C++ structure for each tamed function, containing its parameters and its `tvars` variables. This structure gets an opaque name, discouraging the programmer from accessing it directly.

Programmers are free to use arbitrary C++-stack allocation, as long as no wait points come between the declaration and use of stack-allocated variables. When they do, the underlying C++ compiler generates a warning due to `goto` branches in the emitted code (see the next section).

Maintaining Section 5's R4 requires that each closure know which `rendezvous` it directly contains, so it can cancel them appropriately. This knowledge is unavailable without fully parsing C++: a closure might contain an object of type `foo`, that contains an object of type `bar`, that contains a `rendezvous`, which will in turn share fate with the closure. As a first-order heuristic, Tame marks the beginning and the end of the new closure in memory using simple pointer arithmetic and associates with the closure all `rendezvous` that fall between the two fence posts.

6.2 Entry and Exit Translation

The translation of a tamed function adds to the function one new entry and exit point per `twait` statement. A translated `twait` statement first checks whether a trigger is pending on the corresponding `rendezvous`. If so, control flow continues past the `twait` function as usual; but if not, the function records the current wait point, adds a function pointer for this wait point to the `rendezvous`, and returns to its caller. Later, a trigger on an `event` in the `rendezvous` invokes the recorded function pointer, which forces control to reenter the function and jump directly to the recorded wait point. The Tame translation shifted the function's parameters and safe local variables to a closure structure, so the function can access these values even after reentry.

The Tame preprocessor adds an extra "closure pointer" parameter to each `tamed` function. The closure pointer is null when the function is called normally, causing the translated function body to allocate and initialize a new closure. The closure pointer is non-null when the function is reentered at a later wait point. The names of parameters and safe local variables are changed to opaque identifiers to hide them from the function body; instead, local variables with reference types make these names point into the closure. This strategy reduces the extent to which Tame must understand C++ name lookup, since the translation preserves the function implementation's original namespace. Multiple entry points are simulated with a switch statement at the beginning of the function; each case in the switch jumps to a different label in the function. There is one label for the initial function entry and one for each wait point.

Internally, `mkevent` is a macro that fetches some specially named variables (such as the current closure, or the current implicit `rendezvous` in the case of a `twait` environment). An input of `mkevent(rv, w, t1, t2)` generates a call of the form:

```
_mkevent(__cls, rv, w, t1, t2);
```

for some closure `__cls`. `_mkevent` heap-allocates a new `event` object, packing it with references to all of the supplied arguments. The resulting `event` object provides one method, `trigger`, which takes trigger value parameters with the types of `t1` and `t2`. All of these operations are type-safe through use of C++ templates.

Putting these pieces together, the translation of:

```
1  tamed A::f(int x) {
2    tvars { rendezvous<> r; }
3    a(mkevent(r)); twait(r); b(); }
```

looks approximately like:

```
1  void A::f(int __tame_x, A_f_closure *__cls) {
2    if (__cls == 0)
3      __cls = new A__f__closure(this, &A::fn, __tame_x);
4    assert(this == __cls->this_A);
5    int &x = __cls->x;
6    rendezvous<> &r = __cls->r;
7    switch (__cls->entry_point) {
8    case 0: // original entry
9      goto __A__f__entry__0;
10   case 1: // reentry after first twait
11     goto __A__f__entry__1; }
12 __A__f__entry__0:
13   a(_mkevent(__cls,r));
14   if (!r.has_queued_trigger()) {
15     __cls->entry_point = 1;
16     r.set_reenter_closure(__cls);
17     return; }
18 __A__f__entry__1:
19   b();
20 }
```

Lines 5–6 set up the function body so that references to `x` and `r` refer to closure-resident values. Lines 7–11 direct traffic as it enters and reenters the function after `twait` points. Lines 12–19 are the translation of the user code. Lines 14–17 are the translation of the `twait(r)` call from the original function. If no trigger is queued on `r`, the translation bumps the entry point (line 15) and tells the `rendezvous` to reenter `__cls` via the method `A::fn` when a trigger arrives (line 16). Once that happens, `f` will jump to entry point 1 (line 18) and call `b()`.

6.3 Backwards Compatibility

Our implementation of Tame borrows its event loop and event objects from the *libasync* event library [21]. The key compatibility feature is to implement `events` as *libasync* `callbacks`, allowing legacy functions to interface with tamed functions, and consequently, legacy projects to incrementally switch over to tamed code.

The Tame prototype implements thread support with the Gnu PTH library [12]. PTH supplies stubs for blocking network calls such as `select`, `read` and `write`. Thus the `select` call in *libasync*'s select loop transparently becomes a call to PTH's scheduler. Similarly, blocking network calls in third party libraries like `libmysqlclient` drop into the scheduler and later resume when the operation completes. We also had to make *libasync* call a modified, Tame-aware `select`. This `select` returns early when another thread in the same process triggers an event that should wake up the current thread (something that never happens in single-threaded Tame).

7 Experience With Tame

Like any other expressive synchronization system, Tame requires some mental readjustment and ramp-up time. In most cases, developers need only the `twait{}` environment, which is designed to be simple to learn and comparable to thread programming. With only this subset of Tame, programmers become much more productive relative to vanilla-event coders, and hopefully as productive as thread programmers.

7.1 Web Server

The latest version of OKWS [18], a lightweight Web server for dynamic Web content, uses the Tame system. Its most obvious applications are serial chains of asynchronous function calls, such as startup sequences that involve IPC across cooperating processes. These chains are common in OKWS; Tame lets them occupy a single function body, making the code easier to read.

A more specialized Tame application is in OKWS's templating system, which allows OKWS Web developers to separate their application logic from the HTML presentation layer. In a manner similar to Flash [24], OKWS uses blocking helper processes to read templates from the file system; the main server calls the helper processes asynchronously. However, since templates can be arbitrarily nested, reading one template may require many helper calls. The previous version of OKWS, written without Tame, sacrificed expressiveness for programmability. Web site developers had to request all template files they would ever need when their Web service started up, so that a call to publishing a template in response to a Web request would not block and force a stack rip. In the new version of OKWS, publishing a template is an asynchronous operation, and site developers can therefore publish any file in the `htdocs` directory, at any time. Tame saves developers from the stack ripping problem that previously discouraged this feature.

7.2 An Event-Based Web Site

OkCupid.com [16] is a dating Web site that uses OKWS as its Web server. For several years, its programmers wrote code in the *libasync* idiom to manage concurrency, but in early 2006 switched over to Tame to simplify debugging and to improve productivity. Currently seven programmers, none of whom are the authors, depend on Tame for maintaining and developing site features. The system is easy enough to use so that the first programming project new employees receive is to convert code from the old event-based system to Tame syntax.

OkCupid.com has found Tame's parallel dispatch particularly useful when programming a Web site. When a user logs into the site, the front-end Web logic requests data from multiple databases to reconstruct the user's preferences and server-resident state. To minimize client-perceived latency from disk accesses, these queries can happen in parallel. With just *libasync* primitives, parallelism was hidden in stack-ripped code and caused bugs. Tame's solution is the parallelism inherent in the `twait` environment. To call f and g in parallel, then call h once they both complete, a Tame programmer simply writes:

```
1    twait { f(mkevent()); g(mkevent()); }
2    twait { h(mkevent()); }
```

7.3 An NFS Server

A graduate distributed systems class at MIT requires its students to write a simple Frangipani-inspired [35] file server that implements the NFS Version 3 protocol [4]. In spring 2006, the students had the option to write their assignment with the Tame tool. Four out of 22 students used Tame, most successfully on the source file that implements the file system semantics (about two thousand lines long). Consider, for example, the CREATE RPC, for creating a new file on the server. When given this RPC, the server must acquire a lock, lookup a file handle for the target directory, read the contents of the directory, write out new directory contents, then write out the file, and finally release the lock, all the while checking for various error conditions. The solutions with legacy *libasync* involve code split up over no fewer than five functions, with a stack rip at every blocking point. Students who used Tame accomplished the same semantics with just one function. Quantitatively, the students who used Tame wrote 20% less code in their source files, and 50% less code in their header files. Qualitatively, the students had positive comments about the Tame system and semantics, and strongly preferred writing in the Tame idiom to writing *libasync* code directly.

7.4 Debugging

Tame's preprocessor implements source-code line translation, so debuggers and compilers point the programmer to the line of code in the original Tame input file. The programmer need only examine or debug autogenerated code when Tame itself has a bug. Programmers can disable line-translation and view human-readable output from the Tame preprocessor. Relative to a tamed function in the input file, a tamed function in the output file differs only in its Tame-generated preamble, at `twait` points, and at `return` statements. The rest of the code is passed through untouched.

Tame also has debugging advantages over legacy *libasync* with unmodified debugging tools. With legacy *libasync*, a developer must set a breakpoint at every stack rip point. With Tame, a logical operation once again fits inside a single function body. As a result, a programmer sets a break point at the suspect function, and can trace execution until a blocking point (i.e., `twait`). After the blocking point, control returns to the same breakpoint at the top of the same function, and then jumps to the code directly after the `twait` statement.

Future work calls for a Tame debugger and profiler. In both cases, the runtime nesting of closures is Tame's analogue of the call stack in a threaded program. Slight debugger modifications could allow walking this graph to produce a "stacktrace"-like feature, and similarly, measuring closure lifetime can yield a `gprof`-style output for understanding which parts of a program induce latency. Even under the status quo, programmers can access safe local variables in debuggers by simply examining the members of a function's closure and can walk the closure-chain manually if desired.

7.5 Locks and Synchronization

Programmers using events or cooperative threading often falsely convince themselves that they have "synchronization for free." This is not always the case. Global data on one side of a yield or block point might look different on the other side, if another part of the program manipulated that data in between. With threaded Tame programs, or threaded programs in general, any function invocation can result in a yield, hiding concurrency assumptions deep in the call stack. In practice, a programmer cannot know automatically know when to protect global data structures [1]. Event-only Tame programs make concurrency assumptions explicit, since they never yield; they just return to the main event loop (allowing other computations to run) on either side of a `twait` statement or environment.

When Tame programs require atomicity guarantees on either side of a `twait` (or yield in the case of threads), they can use a simple lock implementation based on Tame primitives. A basic lock class exposes the methods:

```
tamed lock::acquire(event<> done);
void lock::release();
```

The `acquire` method checks the lock to see if it's currently acquired; if so, it queues the given `event`, and if not, it triggers `done` immediately. The `release` method either triggers the head of the `event` queue, or marks the lock as available if no `events` were queued. An example critical section in Tame now looks like:

```
1  tamed global_data_accessor() {
2      twait { global_lock->acquire(mkevent()); }
3      ... touch global state, possibly blocking ...
4      global_lock->release();
5  }
```

We have also built shared read locks with Tame, in which a writer's release of a lock can cause all queued readers to unblock.

8 Performance Measurements

The Tame implementation introduces potential performance costs relative to threaded code and traditional event-driven software. Unlike cooperative-threaded code, and more so than traditional event libraries (e.g. *libasync*), Tame makes heavy use of heap-allocated data structures, such as closures and one-time events. Tame also uses synchronization primitives (namely `rendezvous` and `events`) that are potentially costlier than the lower level primitives in threading packages or *libasync*. We investigate the end-to-end cost of Tame relative to a comparable

	Capriccio	Tame
Throughput (connections/sec)	28,318	28,457
Number of threads	350	1
Physical memory (kB)	6,560	2,156
Virtual memory (kB)	49,517	10,740

Figure 7: Measurements of Knot at maximum throughput. Throughput is averaged over the whole one-minute run. Memory readings are taken after the warm-up period, as reported by `ps`.

high-performance system, and conclude that Tame incurs no performance penalties and makes better use of memory.

8.1 End-to-End Performance

A logical point of comparison for the Tame system is the Capriccio thread package [38]. Like Tame, it provides automatic memory management and cooperative task management; it is also engineered for high performance. The Capriccio work focuses its measurements on the simple "Knot" web server. We compared the performance of the original Capriccio Knot server with a lightly modified, tamed version of Knot. In selecting a workload, we factored out the subtleties of disk I/O and scheduling that other work has addressed in detail [25] and focused on memory and CPU use. We ran a SpecWeb-like benchmark but used only the smallest files in the dataset, making the workload entirely cacheable and avoiding link saturation.

For all experiments, the server was a 2-CPU 2.33 GHz Xeon 5140 with 4GB memory, running Ubuntu Linux with kernel 2.6.17-10, code compiled with GCC version 4.1.2, optimization level `-O2`. Because Capriccio does not compile with more recent compilers, it was compiled GCC version 3.3.5. Glibc and NPTL were both version 2.4. Though the machine has four cores, only one was needed in our experiments (neither Tame nor the other systems tested use multiple CPUs). Tame supports Linux's `epoll`, but its event loop was configured to use `select` in our benchmarks. Capriccio uses the similar `poll` call in its loop. We used an array of six clients connected through a gigabit switch, each making 200 simultaneous requests to the server. The servers were given a thirty-second "warm-up" time in which they pulled all of the necessary files from disk into cache, and then ran for a one-minute test. The results are shown in Figure 7.

The high level outcome is that under this workload, the Capriccio and Tame versions of Knot achieve the same throughput, but Tame Knot uses one-third the physical memory, and one-fifth the virtual memory. We note that neither Knot server in this scenario ever blocks: both servers use 100% of available CPU, even when idle. A version of the Tame server that blocks when there is no work to be done achieves a surprising 4,000 fewer connections per second on our benchmark machine. Another important optimization was to avoid dropping into the select loop when outstanding connection attempts could be `accepted` [3]. Microbenchmarks in Section 8.2 show the select loop is expensive relative to other Tame primitives.

Operation	Min	Median	Mean
Simple function call	63	63	66
Simple function call with `int` allocation	182	196	196
Tame call (`nullfn()`)	399	455	463
`wrap()`	217	224	231
`gettimeofday()`	2618	2660	2781

Figure 8: Cost of system calls and *libasync* and Tame primitive operations, measured in cycles.

Optimizing Capriccio Knot's performance required manual tuning. The size of the thread pool must be sufficiently large (about 350 threads) before Capriccio Knot can achieve maximal throughput. Threads' stack sizes must also be set correctly—stacks that are too small risk overflow, while stacks that are too big waste virtual memory—but the default 128 kB per stack sufficed for these experiments. Capriccio can automate these parameter settings, but the Knot server in the Capriccio release does not use automatic stack sizing, and manual thread settings were more stable in our tests. Further work could bring Capriccio's memory usage more in line with Tame's, but we note that Tame achieves its memory usage automatically without changes to the base compiler.

Memory allocation in Tame Knot happens mainly on the heap, in the form of event and closure allocations. In our test cases, we noted 12 closure allocations and 12.6 event allocations per connection served. We experimented with "recycling" events of common types (such as `event<>`s) rather than allocating and freeing them each time. Such optimizations had little impact on performance, suggesting Linux's malloc automatically optimizes Tame's memory access pattern.

8.2 Microbenchmarks

We performed microbenchmarks to get a better sense for how Tame was spending its cycles in the web benchmark, and to provide baseline statistics for other applications. A first cost of Tame relative to thread programming is closure allocation. We measured closure costs with the most basic tamed function that uses a closure:

```
1  static tamed nullfn()
2    { tvars { int i(0); } i++; }
```

For comparison, we also measure a trivial function, a function that performs a small heap allocation, *libasync*'s closure-approximating function (i.e., `wrap`), and a trivial system call (`gettimeofday`). In each case, an experiment consisted of executing the primitive 10,000 times, bracketed by cycle counter checks. We ran each experiment 10 times and report averaged results over the 10 experiments, and the median results over all 100,000 calls. In all cases, the standard deviation over the 10 experiments was within 5% of the mean. Figure 8 summarizes our results: entering

a tamed function is about 2.2 times the cost of a simple function with heap allocation, and 1.8 times the cost of a `wrap` invocation.

A second function, `benchfn`, measures Tame overhead when managing control flow:

```
1  static tamed benchfn (int niter, event<> done) {
2     tvars { int i; }
3     for (i = 0; i < niter; i++)
4        twait { timer(0, mkevent()); }
5     done.trigger();
6  }
```

Line 4 of `benchfn` is performing Tame's version of a thread fork and join. A call to `mkevent` and later `twait` is required to launch a potentially blocking network operation, and to harvest its result. Unlike a *libasync* version of `benchfn`, the tamed version must manage closures, an implicit `rendezvous`, and jumping into and out of the function once per iteration. We compare three versions of `benchfn`: with an implicit `rendezvous`, with an explicit `rendezvous`, and with only *libasync* features.

We ran all versions with `niter=100`, and repeated the experiment one thousand times. The results are presented in Figure 9. All experiments spend a majority of cycles in the core select loop. The `benchfn` that uses an implicit `rendezvous` is only slightly more expensive, performing within 2% of the native *libasync* code. Tame's low-level implementation special-cases the implicit `rendezvous`, reducing memory allocations and virtual method calls along the critical path. Hence, the `benchfn` version that uses an explicit `rendezvous` runs about 6% slower still. We also experimented with replacing *libasync*'s native scheduler with that of the PTH thread library, as is required when running Tame with thread support.

Based on these benchmarks, we can estimate how Tame Knot's CPU time is spent. Tame Knot uses 81,877 cycles for each request. Assuming the microbenchmark results hold, and given Tame Knot's use of 12.6 events and 12 closure allocations per request, roughly 7.6% of these cycles are spent on event management and 6.8% on closure management, with the remainder going towards system calls and application-level processing.

Figure 9 also gives similar benchmarks for a version of `benchfn` written in pure thread abstractions,

```
1  static void noop() { pthread_exit(NULL); }
2  void benchfn_thr(int niter) {
3     for (int i = 0; i < niter; i++) {
4        pthread_t t;
5        pthread_create(&t, NULL, noop, NULL);
6        pthread_join(t, NULL);
7     }
8  }
```

and a version using Tame's thread wrappers:

```
1  static void noop_tame() {}
```

Function	Cycles	In Core	Outside
benchfn without Tame	5,251	4,906	344
benchfn with `twait{}`	5,331	4,840	491
with PTH core loop	6,010	5,476	534
benchfn with `twait(r);`	5,642	4,887	755
with PTH core loop	6,565	5,678	887
benchfn_thr in PTH	37,540	-	-
in NPTL	28,803	-	-
in Capriccio	7,892	-	-
benchfn_tame_thr	64,957	-	-

Figure 9: Results from running the `benchfn` code in 1,000 experiments, with `niter=100`. Costs shown are the average cycles per iteration, averaged over all experiments. These costs are broken down into cycles spent in the core event loop, and time spent outside.

```
2  void benchfn_tame_thr(int niter) {
3     for (int i = 0; i < niter; i++)
4        twait { tfork(wrap(noop_tame)); }
5  }
```

A thread allocation and join is five to seven times as expensive as an event allocation and join in Tame when using standard Linux libraries like PTH and NPTL. Tame's thread wrappers added additional overhead relative to native PTH since they require locks and condition variables. Capriccio is much faster and competitive with Tame. Traditionally, threaded programs allocate threads not quite as cavalierly as `benchfn_thr`; they might use thread-pooling techniques to accomplish more than one operation per thread. However, examples like those in Sections 3.2 and 3.3 and those in real-world Web site programming (Section 7.2) benefit greatly from repeated thread creation and destruction. Tame primitives are certainly fast enough to support this.

In sum, Tame's primitive operations are marginally more expensive than *libasync*'s and roughly equivalent to those of a good thread package. The observed costs are cheap relative to real workloads in network applications.

9 Summary

Tame confers much of the readability advantage of threads while preserving the flexibility of events, and modern thread packages have good performance: the clichéd performance/readability distinction between events and threads no longer holds. Programmers should choose the abstraction that best meets their needs. We argue that event programming with Tame is a good fit for networked and distributed systems. The Tame system has found adoption in real event-based systems, and the results are encouraging: fewer lines of code, simplified memory management, and simplified code maintenance. Our hope is that Tame can solve the software maintenance problems that plague current event-based systems, while making events palatable to a wider audience of developers.

ACKNOWLEDGMENTS

We thank the following people for helpful comments on the Tame system and earlier drafts of this paper: Jeff Fischer, Jon Howell, Rupak Majumdar, Todd Millstein, Michael Walfish, Russ Cox, Jeremy Stribling, the other members of the PDOS group, and the anonymous reviewers. Russ Cox and Tim Brecht helped us in benchmarking Capriccio, and we thank Vivek Pai for his "flexiclient" workload generator. Chris Coyne inspired the authors to think about simplifying events. Thanks to him and the developers at OkCupid.com who have adopted the system. We also thank David Mazières for creating the *libasync* system on which Tame is based. Eddie Kohler's work on Tame was supported by the National Science Foundation under Grant No. 0427202.

The Tame system (with minor syntactic differences) is distributed along with the *libasync* libraries, all under a GPL version 2 license, at `http://www.okws.org/`.

REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proc. 2002 USENIX Annual Tech. Conference*, June 2002.

[2] A. Birkett. Parsing C++. `http://www.nobugs.org/developer/parsingcpp`.

[3] T. Brecht, D. Pariag, and L. Gammo. accept()able strategies for improving Web server performance. In *Proc. 2004 USENIX Annual Tech. Conference*, June 2004.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[5] K. Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3), May 1999.

[6] R. Cunningham and E. Kohler. Making events less slippery with eel. In *Proc. HotOS-X*, June 2005.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th SOSP*, October 2001.

[8] G. L. Davies. Teaching concurrent programming with Pascal-FC. *SIGCSE Bulletin*, 22(2), 1990.

[9] U. Drepper. The native POSIX thread library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`.

[10] T. Duff. Duff's device. `http://www.lysator.liu.se/c/duffs-device.html`.

[11] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proc. 2006 SenSys*, Nov. 2006.

[12] R. S. Engelschall. Portable multithreading: The signal stack trick for user-space thread creation. In *Proc. 2000 USENIX Annual Tech. Conference*, June 2000.

[13] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. 1st NSDI*, March 2004.

[14] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10), 1974.

[15] P. Hudak et al. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5), 1992.

[16] Humor Rainbow, Inc. OkCupid.com. `http://www.okcupid.com`.

[17] G. Jones. *Programming in Occam.* Prentice Hall International (UK) Ltd., 1986.

[18] M. Krohn. Building secure high-performance web services with OKWS. In *Proc. 2004 USENIX Annual Tech. Conference*, June 2004.

[19] J. Lemon. Kqueue: A generic and scalable event notification facility. In *Proc. 2001 FREENIX*, 2001.

[20] P. Li and S. Zdancewic. Combining events and threads for scalable network services. In *Proc. 2007 PLDI*, Jun 2007.

[21] D. Mazières. A toolkit for user-level file systems. In *Proc. 2001 USENIX Annual Tech. Conference*, June 2001.

[22] Microsoft. Inside I/O completion ports. `http://www.microsoft.com/technet/sysinternals/information/IoCompletionPorts.mspx`.

[23] MySQL AB. MySQL. `http://www.mysql.com`.

[24] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proc. 1999 USENIX Annual Tech. Conference*, June 1999.

[25] D. Pariag, T. Brecht, A. Harji, P. Buhr, and A. Shukla. Comparing the performance of web server architectures. In *Proc. 2007 EuroSys*, March 2007.

[26] K. Park and V. S. Pai. Connection conditioning: Architecture-independent support for simple, robust servers. In *Proc. 2006 NSDI*, May 2006.

[27] N. Provos. libevent —an event notification library. `http://www.monkey.org/~provos/libevent`.

[28] N. Provos. A virtual honeypot framework. In *Proc. 13th USENIX Security Symposium*, Aug 2004.

[29] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogeneous architectures. In *Proc. 27th International Symposium on Fault-Tolerant Computing*, June 1997.

[30] J. H. Reppy. CML: A higher concurrent language. In *Proc. 1991 PLDI*, June 1991.

[31] Silicon Graphics, Inc. State threads for Internet applications. `http://state-threads.sourceforge.net/docs/st.html`.

[32] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2), 1998.

[33] G. Steele. *Common Lisp.* Digital Press, Jun 1984.

[34] J. Stribling, J. Li, I. G. Councill, M. F. Kaashoek, and R. Morris. OverCite: A distributed, cooperative CiteSeer. In *Proc. 2006 NSDI*, May 2006.

[35] C. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proc. 16th SOSP*, October 1997.

[36] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proc. 2006 NSDI*, May 2006.

[37] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proc. HotOS-IX*, May 2003.

[38] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for Internet services. In *Proc. 19th SOSP*, October 2003.

[39] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS Defense by Offense. In *Proc. 2006 NSDI*, May 2006.

[40] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th SOSP*, October 2001.

[41] N. Zeldovich et al. Multiprocessor support for event-driven programs. In *Proc. 2003 USENIX Annual Tech. Conference*, June 2003.

MapJAX: Data Structure Abstractions for Asynchronous Web Applications

Daniel S. Myers
MIT CSAIL

Jennifer N. Carlisle
MIT CSAIL

James A. Cowling
MIT CSAIL

Barbara H. Liskov
MIT CSAIL

Abstract

The current approach to developing rich, interactive web applications relies on asynchronous RPCs (Remote Procedure Calls) to fetch new data to be displayed by the client. We argue that for the majority of web applications, this RPC-based model is not the correct abstraction: it forces programmers to use an awkward continuation-passing style of programming and to expend too much effort manually transferring data. We propose a new programming model, MapJAX, to remedy these problems. MapJAX provides the abstraction of data structures shared between the browser and the server, based on the familiar primitives of objects, locks, and threads. MapJAX also provides additional features (parallel **for** loops and prefetching) that help developers minimize response times in their applications. MapJAX thus allows developers to focus on what they do best–writing compelling applications–rather than worrying about systems issues of data transfer and callback management.

We describe the design and implementation of the MapJAX framework and show its use in three prototypical web applications: a mapping application, an email client, and a search-autocomplete application. We evaluate the performance of these applications under realistic Internet latency and bandwidth constraints and find that the unoptimized MapJAX versions perform comparably to the standard AJAX versions, while MapJAX performance optimizations can dramatically improve performance, by close to a factor of 2 relative to non-MapJAX code in some cases.

1 Introduction

> *"It is really, really, really hard to build something like Gmail and Google Maps," said David Mendels, general manager of platform products for Macromedia. "Google hired rocket scientists... Most companies can't go and repeat what Google has done." [1]*

Recent months have shown an explosive growth in rich, interactive content on the World Wide Web — a phenomenon termed *Web 2.0*. Central to this growth is a communication technique known as AJAX: Asynchronous Javascript and XML [2]. Before AJAX, web applications were forced to fetch an entire page from the server in order to display any new data. By contrast, AJAX allows Javascript programs to send requests to the server without reloading the page or blocking the user interface. This has permitted the development of a new class of highly-responsive, desktop-like applications on the web. Moreover, support for the underlying AJAX mechanisms is ubiquitous, having been present in web browsers since the late 1990s, so these applications can be delivered without the need for third-party plugins.

The current AJAX programming model has two significant shortcomings, however. First, AJAX requires that web clients request content from web servers using asynchronous HTTP requests: the client bundles any statements that depend on the result of the request into a callback that will be executed when the response to the HTTP request arrives. This approach forces programmers to use an awkward continuation-passing programming style and thread program state through a series of callback functions. While various toolkits [9, 15] elevate the level of abstraction to that of an RPC, none has eliminated the use of continuations and callbacks.

Additionally, a programmer using AJAX must develop his or her own techniques for avoiding delays associated with fetching content from the server. These delays can be reduced by prefetching content before it is needed, and by sending requests in parallel. Neither AJAX nor current tools built on top of it provides direct support for these approaches.

This paper presents MapJAX, a data-centric framework for building AJAX applications without any new client-side software. In place of asynchronous RPCs, MapJAX provides the programmer with the illusion of logical data structures shared between the client and server. These data structures appear to be ordinary objects that can be accessed through normal Javascript method calls, hiding the underlying complexity of continuations and callback functions. Instead, the code that causes a fetch of data from the server can be thought of as running in its own thread. The thread blocks while the call is being processed and then continues running when the server response arrives. The MapJAX approach thus allows users to create programs with the mental model to

which they have grown accustomed: threads and objects.

In addition, MapJAX also provides a number of mechanisms that enable efficient interaction between client and server. Given that the cost of communication far exceeds the cost of computation in this setting, we focus on ways to reduce communication delays. First, MapJAX allows application programmers to decrease the latency involved in data structure access by using spare bandwidth to prefetch objects. The MapJAX runtime maintains a cache of previously fetched objects and avoids communication delay when the requested content is present in the cache.

Second, MapJAX provides a mechanism that allows a number of fetches to be sent in parallel. This is provided in the form of a parallel **for** statement. A common use case for web applications is to copy a range of elements to the screen. This is naturally expressed as a sequential **for** loop over a shared data structure, albeit with poor performance. Instead, the parallel **for** starts up the iterations in parallel, allowing the requests to be issued immediately and concurrently.

In order to allow applications to express ordering constraints on these loops while preserving concurrency, MapJAX provides a new locking mechanism that allows a thread to reserve a lock in advance of acquiring it. Lock reservation is non-blocking and simply places the identifier of the thread into the lock queue. Later, the thread acquires the lock using an acquisition function which blocks until the lock is available. As detailed in Section 3.5, this model allows threads to generate requests in parallel, yet process their responses in order.

MapJAX also provides a few additional features to increase the effectiveness of prefetching and parallel **for**. It is able to group a number of requests into one communication with the server, and it allows the program to cancel requests that are no longer needed.

MapJAX is defined as a small veneer over Javascript, as a goal of the system is to require a minimum of programmer retraining. Its implementation consists of a compiler that translates MapJAX programs to Javascript, and a runtime system written in Javascript that provides the needed support for the MapJAX features. Despite significant work on programming environments for web applications [13, 10, 9, 3, 15, 4, 11], we are not aware of any existing work that provides a programming model like that of MapJAX.

We have used MapJAX to implement three prototypical web applications–a mapping application, an email client, and a search autocomplete application. These were all implemented with relative ease, while benefiting from prefetching and parallel **for** loops. Our results show that MapJAX has minimal overhead compared to an equivalent AJAX program. The results also show that use of our advanced features resulted in substantial performance improvements. to as much as a factor of 2 in some cases.

2 Javascript and AJAX Applications

We first provide a brief overview of Javascript and AJAX. The Javascript language was originally developed at Netscape in 1996 to allow interactivity in web pages. In particular, Javascript allows handlers to be registered for activation in response to various events that can occur on the page, such as the page being loaded, the user clicking on a link, moving his or her mouse over an image, and so forth. While Javascript has seen use outside of the context of web pages, within this context Javascript programs are event-based. As only one event handler can execute concurrently and scheduling is non-preemptive, Javascript programs can be viewed as executing under a single-process, event driven (SPED) model similar to e.g. Zeus [17].

Application programmers can modify web page content from Javascript using the Document Object Model (DOM), a programmatically-accessible, tree-structured representation of the elements on the page.

Until relatively recently, Javascript was used only for purely client-side tasks, such as verifying text input into a zip-code form field. Since the late 1990's however, Javascript has contained an "XMLHttpRequest" object, allowing programmers to send asynchronously-handled HTTP requests for XML-formatted data. The realization in the web development community that this object could be used to fetch new data without reloading a page gave rise to AJAX, standing for "Asynchronous Javascript and XML", although more recently alternative encodings such as Javascript Object Notation (JSON) [5] have been used in place of XML.

3 Programming Model

This section describes the MapJAX programming model. We begin with the basic features that allow programmers to write working programs (shared data structures and non-preemptive threads), then describe various features that help them to improve performance (data prefetching, parallel **for** loops, RLU locks, and request canceling).

3.1 MapJAX Maps

MapJAX objects represent logical data structures shared between the client and server, and are at the core of the MapJAX system. These objects are collections of elements, e.g., a collection of email messages, or a collection of grid cells in a mapping application. Each map is associated with a URL at the server to which requests are sent to retrieve elements.

- *Cmap(String url, Object prefetch, String transport).* Creates a new MapJAX map, where *url* is the URL that will be used to fetch its elements, *prefetch* is the prefetching policy, and *transport* identifies the type of transport used to make requests.
- *Object access(String key).* Returns the element named by *key*. Alternatively, returns a failure code if the *key* is invalid or the network fails.
- *Boolean has(String key).* Returns true if the element named by *key* is present in the cache.
- *void prefetch(String[] keys).* User-initiated prefetching.
- *void cancel(String key).* Cancel an outstanding access.

Figure 1: MapJAX maps API.

The base representation provided by MapJAX for collections of elements is a map from keys to values, where the keys are strings, and the values can be any Javascript data type (we assume that maps do not contain other maps). Maps are supported at the server by this base representation. At the client side, however, it can be useful to access the shared structure at a higher level of abstraction, such as an array or tree.

MapJAX provides three higher-level abstractions: one and two-dimensional arrays, and trees, with the class names ARRAYMAP, GRIDMAP, and TREEMAP, respectively. One dimensional arrays use integers as keys, two dimensional arrays use pairs of integers as keys, and trees use strings where each string represents the path from the root to the element of interest. Programmers are free to implement their own abstractions using any of these primitives, and MapJAX is capable of supporting general object graphs.

MapJAX objects appear as ordinary Javascript objects. Figure 1 shows the interface to MapJAX maps; although Javascript is not statically typed, we have included types in the method descriptions to clarify the presentation.

The constructor takes as an argument the URL to be used to fetch elements of the map. It also takes a *prefetch* object, which defines how prefetching works for this map; prefetching is described in Section 3.3. The third argument specifies the type of transport to be used to retrieve elements; we defer discussion of this implementation detail to Section 4.

The most interesting method of a MapJAX object is *access*. A call of this method, e.g., *foo.access("bar")*, is a blocking call: it will not return until the requested element has been retrieved from the server, although it may return immediately if the value is already cached. The MapJAX threading model (discussed below) allows other Javascript or browser code to execute while the call is blocked.

The *has*, *prefetch*, and *cancel* methods are used for cache management and server functionality; they are discussed in later sections.

Maps are read-only. Given the generally read-oriented nature of the web, we do not feel this to be an overly onerous limitation, but leave write-enabled structures as an area of future work.

3.2 Threads

A MapJAX program consists of a collection of *threads*. A new thread is created each time an event handler is invoked by the browser. Threads are also created for iterations in the parallel **for** as discussed in Section 3.4.

Threads are scheduled non-preemptively: a thread retains control until relinquishing it. A thread relinquishes control when it finishes processing an event or when it makes a blocking call on a MapJAX object. For example, when a thread calls the *access* method of a MapJAX map object, causing an RPC to be made to the server, it relinquishes control; it will regain control at some point after the result of the RPC arrives (or the RPC is aborted because of communication problems).

Threads are implicit at present, and relinquishing control is also implicit. However, it would not be difficult to extend MapJAX to support explicit threads and thread management (e.g., a **yield** statement) if this turned out to be useful.

It is worth pointing out that the concurrency in MapJAX programs also exists in AJAX programs. The difference is that in MapJAX programmers can think of each event handler as running in its own thread, with the system switching control among threads automatically, whereas in AJAX, the programmer needs to write callbacks and continuation functions.

3.3 Prefetching

All MapJAX maps support prefetching via programmer-defined prefetching policies. A prefetching policy is a Javascript object. It provides a method, *getPrefetchSet*, that, given a key, returns a set of keys that identify elements to prefetch. Prefetching policies are usually specialized to the kind of MapJAX map in use in the web application. E.g., for a map-viewing application, the prefetching policy might indicate to fetch all grid cells adjacent to the one being requested. Additionally, the prefetching policy can be tailored to a particular higher level abstraction; e.g., one defined for arrays would expect keys to be integers.

Figure 2 provides an example prefetching object that implements a "read-ahead K" policy for an ARRAYMAP.

```
// Javascript object constructor
function ReadAheadKPolicy(k) {
    this.k = k;
}
// Javascript object definition
ReadAheadPolicy.prototype = {
    getPrefetchSet: function(idx) {
        var pf_set = new Array();
        for (var i = 1; i <= k; ++i) {
            pf_set.push(idx + k);
        }
        return pf_set;
    }
}
```

Figure 2: Example read-ahead-k prefetching policy for an ARRAYMAP.

The *getPrefetchSet* method merely identifies elements of interest. The MapJAX runtime ensures that elements already present in the cache will not be refetched, so these policies do not need to be aware of the state of the cache.

Calls to *getPrefetchSet* are made automatically as part of processing a call to the *access* method. *Access* calls *getPrefetchSet* on the prefetch policy object associated with the map and then requests a fetch of the original key plus all the keys returned by the call. A programmer can also initiate ad-hoc prefetching by calling the *prefetch* method of a map object with an array of keys to prefetch. This method informs the MapJAX runtime of the need to prefetch the elements and then returns immediately (it is non-blocking). The actual fetching occurs in the background.

Custom prefetch policies can be written with a minimal amount of effort from application programmers, allowing prefetching to be tailored to specific web applications. We note additionally that these policies need not be static: as full-fledged Javascript objects, they can maintain internal state to adapt based on the request history.

3.4 Parallel for Loops

A common use case for web applications is to copy a range of elements to the display. The obvious way to program this is to use a **for** loop, where each iteration is responsible for fetching and rendering each element. A sequential execution model is not well suited for the processing of such a loop, however, since it will force one iteration to complete before the next iteration begins; in particular, this will needlessly delay the launch of RPCs to fetch missing data. Prefetching helps but does not completely solve the problem.

To optimize this common and important case, we introduce a parallel **for** statement into the MapJAX language, written **pfor**. The semantics of this statement are as follows. Each iteration runs in a separate thread. Control starts with the first loop iteration; as soon as it blocks, the next iteration starts to run, and so on. More formally, we guarantee that each iteration will initially be given a chance to run in loop order; after the thread corresponding to that iteration yields control, however, it regains control in an arbitrary order with respect to other threads in the loop. Locks, described in the next section, can be used to impose additional ordering constraints if need be. Control passes to code following the loop only once all the iteration threads have terminated.

Our parallel **for** loops are thus similar to standard parallel **for** loops in that they require loop iterations not to effect the termination condition of the loop. They differ slightly, however, because they explicitly start iterations in loop order. Combined with our novel locks, discussed below, this allows programmers to enforce useful ordering constraints that could not be captured with a standard parallel **for**.

The use of the parallel **for** statement can provide considerable performance benefits, as discussed further in Section 6.

3.5 Locks

Any language with concurrency requires some mechanism for its control. In MapJAX, we provide programmers with a novel type of local lock. These locks can be used in the normal way: first a thread acquires the lock and later releases it. However, our locks also provide the ability to reserve the lock in advance of acquiring it. We call these RLU locks because of the "reserve/lock/unlock" regime for using them.

Reserving a lock doesn't block the thread; instead it records the thread on the end of a reservation list. When a thread executes the *lock* method, it will be delayed until the lock is available *and* it is the earliest thread on the list. The interface for MapJAX locks is given in Figure 3. Note that a thread can call the *lock* method without having previously reserved the lock. In addition, the *unreserve* method can be called to give up a reservation.

RLU locks are motivated in large part by our **pfor** loops. Consider a **pfor** loop in which the programmer intends for each iteration to update a shared variable in loop order using data fetched from the web server through a shared map. While the iterations are started in loop order, the responses to the fetch requests may arrive in a different order.

With normal locks, the only solution would be for each iteration to acquire the lock on the shared variable *before* performing the *access* call, thus preventing the iterations from making their fetch requests in parallel. With RLU locks, threads reserve the lock in loop order. They may

- *RLUlock( )*. Creates a new lock object. The lock is available and the reservation list is empty.
- *void reserve( )*. If the thread is already on the reservation list for the lock object, does nothing. Otherwise adds the thread to the end of the reservation list.
- *void lock( )*. If the thread isn't on the reservation list, adds it to the end of the list. Blocks until this thread is at the front of the list and the lock is available. Then acquires the lock and removes the thread from the list.
- *void unlock( )*. If this thread holds the lock, releases the lock, else does nothing.
- *void unreserve( )*. If this thread is on the reservation list, removes it from the list, else does nothing.

Figure 3: MapJAX RLU locks API.

then immediately call *access* and initiate the transfer of remote data, blocking to acquire the lock only when absolutely necessary (before updating the shared variable). The code is given in Figure 4.

```
var sharedData = new ArrayMap(...);
var l = new RLULock();
pfor(var i = 0; i < 47; ++i) {
  l.reserve( );
  var newData = sharedData.access(i);
  l.lock( );
  localObject += newData;
  l.unlock( );
}
```

Figure 4: Locking example.

Finally, we note that RLU locks are useful outside of a **pfor** statement as well: the network reordering issues they are designed to address can arise any time multiple threads seek to synchronize their accesses to an object, and they can also be used as normal locks.

3.6 Request Canceling

When bandwidth is limited, applications must manage it carefully. Request canceling is one mechanism by which they may do so. Specifically, sometimes an application can determine that certain data elements are no longer needed. For example, in our mapping application, a user scrolling quickly in a low-bandwidth environment can trigger two updates for the same cell on screen, where only the latter update is required. If the application can indicate to the runtime that the first update is no longer needed, considerable bandwidth can be saved in the case where the RPC for the first request has not yet been sent to the server.

Request canceling is supported by the *cancel* method of MapJAX maps. This call takes a key, k, as an argument. If there is no outstanding RPC with k as an argument, the cancel request has no effect. Otherwise, the RPC with k as an argument *might* be canceled; other RPCs might be canceled as well. If an RPC is canceled, any call of *access* that is waiting for the results of that RPC will return with a failure code. The failure code allows the thread that is waiting for the result to act appropriately when it starts running again.

The MapJAX runtime determines what is canceled based on heuristics about the utility of the outstanding RPCs. Generally, if k was an argument to a previous *access* call, the system will cancel all RPCs corresponding to that call, i.e., requests both for k and for other keys identified by prefetching. However it will not cancel an RPC for one of the prefetch keys if it appeared as an argument to a later *access* call or contains requests for non-canceled items. More details about how canceling works can be found in Section 4.2.4. Note that programmers needn't be concerned with these details; instead they simply cancel based on knowledge of what is no longer needed, and the runtime makes the ultimate decision.

4 Implementation

This section describes the implementation of MapJAX. MapJAX is presented to the user as a small extension to the standard Javascript language. The MapJAX implementation has three parts: a compiler that translates MapJAX programs to standard Javascript, a client-side runtime Javascript library that implements the majority of the MapJAX programming model, and a server-side library.

The current version of MapJAX implements the vast majority of the programming model described in Section 3, although it is still an unoptimized prototype. The only features not fully implemented are some request-canceling corner cases described in Section 4.2.4.

4.1 MapJAX Language and Compiler

A major goal for MapJAX is to allow programmers to access data at servers using normal method calls, thus avoiding the complexity of programming with continuations and callbacks. MapJAX also provides support for writing high-performance code, including the parallel **for** statement.

Both blocking calls and the parallel **for** statement require the use of a compiler whose job it is to produce the corresponding Javascript program. This code is based

on callbacks and continuation functions, which permit an efficient implementation.

The MapJAX compiler needs to be able to recognize the features requiring translation. We accomplish this as follows. Blocking method calls are indicated by using special names for these methods: these names always end in "#" (e.g., access#, lock#). The parallel **for** statement is indicated by using an additional keyword **pfor**. Thus our extensions to Javascript are very small. We opted for this approach because we wanted MapJAX to remain similar to Javascript, so that programmers who already knew Javascript would be able to use MapJAX without much effort.

When the compiler encounters one of the blocking method calls, it computes the continuation of the call, packages that code as a continuation function, and adds that function as an extra argument to the call. Additionally, the compiler applies this procedure transitively: any function that calls a blocking method is tagged, and the compiler applies this procedure for any call to a tagged function. Thus, code using the MapJAX programming model is converted into callback-based code compatible with standard Javascript.

The code produced for the **pfor** statement also makes use of continuations and callbacks. Here the compiler must produce code to spawn each iteration as a new thread, to produce the next thread when the previous one blocks, and to ensure that the code after the loop isn't executed until all iterations have terminated.

4.1.1 Function Denesting

Javascript supports nested function declarations, and the initial version of the compiler used them in the generated code: continuation functions were nested in the function from which they were generated, which allowed easy access to variables declared therein. (This would also be an attractive way to write standard AJAX code.) Due to the Firefox implementation of Javascript, however, performance of this code was quite poor: we found that access to variables declared in a nesting hierarchy was considerably slower than access to variables declared in a top-level function. Therefore, the compiler now performs an optimization pass in which the nested code generated by the compiler is fully de-nested; variables needed by formerly-nested functions are stored and passed explicitly in "closure objects." Nested code that existed in the original input file is not denested.

4.2 Client-side Runtime

The majority of MapJAX is implemented in the client-side runtime. Specifically, the runtime provides support for handling accesses to MapJAX objects (including RPC transmission and cache management), creating and scheduling threads of control, and locks.

4.2.1 Object Cache

The MapJAX runtime makes use of a Javascript object that implements a cache for MapJAX object elements. The cache holds previously fetched object elements. Each time a new element arrives from the server, it is added to the cache. Elements are removed from the cache based on TTLs; these TTLs are provided by the server and are sent in the RPC replies. TTLs are intended to ensure data freshness, not to manage the size of the cache. In general, we believe that cache management policies are relatively uninteresting for these applications, given the large size of the cache relative to the amount of data that would normally be downloaded in a reasonable timeframe. Adding sophisticated management policies would be straightforward and able to draw on the large body of existing work.

4.2.2 Accessing Objects

When the *access* method of a MapJAX map is invoked, it first computes the set of prefetch keys. Then it passes the requested key, the set of prefetch keys, and associated callback function (generated by the compiler) to the MapJAX runtime. The runtime interacts with the cache to determine which of the requested elements need to be fetched, and it prunes the list to remove all elements currently present in the cache.

If any elements remain after this pruning, the runtime initiates a request or requests to the server for the missing elements. Then, if the cache contains the requested element, the callback function is invoked immediately. Otherwise, the MapJAX runtime stores the callback function. When an element arrives from the server, callback functions pending on that element are invoked in the order they were submitted.

The MapJAX runtime maintains information about each pending *access* method call: it records the key requested as an argument of that call, plus any additional prefetch keys, the callback, and the RPCs generated to satisfy the request.

4.2.3 Transport Abstraction

The communication protocol is abstracted into a separate class, allowing different transports to be used to fetch objects. For example, AJAX requests through the XMLHttpRequest object can only be used to fetch text data. Binary data, such as images, are not supported. However, by substituting a class that uses the browser's support for loading images directly, rather than AJAX calls, we can support image requests using the same model as

is used for text data. The purpose of the third argument of MapJAX object constructors, left unspecified earlier, is to specify which transport should be used to service misses for that object.

4.2.4 Request Canceling

Request canceling is currently implemented in a simple way; requests are canceled if they contain a single cancelable element. A full implementation would be designed as follows. The MapJAX runtime looks through its list of outstanding *access* requests and their associated RPCs. If the key k being canceled is the key requested by some *access* request, we cancel all RPCs associated with that request except for RPCs requesting keys that are listed in more recent *access* requests. If k is not a requested key (i.e., it is a key generated by a prefetching policy), we cancel the RPC containing the request for it, provided any other keys in the request have already been canceled (the runtime carries out the necessary bookkeeping to determine if they have been).

4.2.5 Request Combining

For performance and scalability reasons, requests to the server for MapJAX object values should be grouped into single messages when possible. The MapJAX runtime cannot predict the future and doesn't know when it receives a request whether it should wait for another. However, when executing the code corresponding to a **pfor** loop, it is clearly advantageous to wait. In this case, MapJAX defers sending any requests until each iteration of the loop has been given a chance to run, offering opportunities for combining.

4.2.6 Callstack Depth Monitoring

Because the client runtime uses continuations, it is possible for the call stack to grow excessively deep. In particular, consider a (non-parallel) **for** loop over an array where all elements are already locally cached. In this case, each loop iteration will add another stack frame, and moderate-sized loops were found to exceed the maximum Javascript stack depth in practice. To cope with this issue, the compiler inserts code to track the depth of the stack at runtime, and the MapJAX runtime monitors this value. If the stack depth grows beyond a given value, the runtime will break the call chain by using Javascript-provided facilities to schedule future event execution (*window.setTimeout*) to execute the next call, rather than allowing it proceed directly.

4.3 Server-side Library

Implementing MapJAX objects requires cooperation from the server. Specifically, each object is associated with a URL on the server that accepts requests for one or more elements in the corresponding shared data structure and returns the corresponding values and TTLs. MapJAX provides a library for use in Java Servlets and Java Server pages for building such servers, but nothing about MapJAX requires the use of Java on the server-side: any software able to process HTTP requests with request parameters will suffice.

5 Applications

We have implemented a number of web applications to evaluate MapJAX based on both programming efficiency and performance. Our chosen applications replicate three prototypically successful AJAX applications: Google Suggest, Google Maps, and Gmail. Each was implemented from scratch using both standard AJAX techniques and MapJAX. Here, we describe the applications and their implementations; in Section 6, we describe their performance under the MapJAX framework.

5.1 Auto-Complete Application

The search auto-complete application, echoing the functionality of Google Suggest, is representative of applications where very low-latency fetching of server data is required. Here, a user types a search phrase into a text box within a web page, and suggested text completions are offered in real-time. A TREEMAP MapJAX object is used to implement a trie providing access to the suggestion set for each successive keypress. Cache misses must be handled with low overhead to ensure responsiveness for typists of even moderate speed.

As each keypress generates a new completion set that obsoletes any previous one, and given that the network may reorder messages, care must be taken to ensure that the correct data are always displayed. MapJAX locks provide a simple mechanism to enforce this constraint: the handling code for a keypress event simply reserves a lock on the completion display object, accesses the shared trie, then locks and updates the display object.

Given the speed at which users type, completion set prefetching can yield a noticeable improvement in application performance. Figure 5 illustrates the expression of a custom prefetching policy in the MapJAX framework. The prediction of the next character that the user will input is based on the last character he or she has entered: we predict that a consonant will follow a vowel, and vice-versa, which is an approximation to English word structure.

```
EnglishPrefetchPolicy.prototype = {
  getPrefetchSet: function(idx) {
    var lastchar = idx.charAt(idx.length - 1);
    var pset = new Array();
    if (this.isVowel(lastchar))
      for (var i=0; i<this.con.length; i++) {
        pset[i] = idx + this.con[i];
      }
    } else {
      for (var i=0; i<this.vowels.length; i++) {
        pset[i] = idx + this.vowels[i];
      }
    }
    return pset;
  }
};
```

Figure 5: Implementation of a custom prefetching policy in MapJAX. This policy uses a basic model of the English language to predict future search queries based on the current query.

5.2 Mapping Application

Our mapping application closely resembles Google Maps: it provides the user with a mobile viewport over a large map. By clicking and dragging the viewport, the user can examine different portions of the map. This example exhibits the implementation of a complex web application (one widely claimed to be difficult to implement) with relatively little effort under the MapJAX framework. The grid nature of the data is well matched by a two-dimensional array abstraction, provided by the MapJAX GRIDMAP. While the logical dimensions of this grid are extremely large (100,000+), MapJAX requires only enough memory on the client to store the data accessed. Moreover, MapJAX is able to stream data as they are needed, rather than requiring the entire working set to be transferred at once.

The map application takes advantage of parallel data fetches from the server and uses locks to ensure that the proper elements are displayed. This application presents a particular challenge, however, because of the bandwidth requirements involved. If prefetching is implemented poorly, it can increase response times by delaying requests that satisfy cache misses. Even if prefetching decreases response times, overly-aggressive prefetching wastes bandwidth, causing the application provider to incur unnecessary bandwidth costs. In our experiments we implement a simple omnidirectional prefetch policy, OMNI, which fetches all tiles within a square of size k of the accessed tile.

When the user moves to another region of the grid, the current set of tiles will be rendered obsolete as new information is requested. Often, if the user is moving quickly through the space, tiles that have been requested for the current display have not yet arrived. Furthermore, these tiles have associated prefetch sets that have also been rendered obsolete. Fetching these unwanted tiles to satisfy outstanding access requests wastes bandwidth. With request cancellation, these requests can be eliminated.

The mapping application is not actually an AJAX application due to the inability of AJAX to transfer binary data. Both the MapJAX and non-MapJAX implementations use native browser support for loading and caching images. The example illustrates the ability of MapJAX to provide a uniform interface to data, regardless of its type.

5.3 Webmail Application

We implemented a two-pane webmail application. The left pane displays a list of email message headers (sender, date, and subject); when one of the headers is clicked, the corresponding message body is displayed in the right-hand pane. The left pane contains at most 40 message headers; additional message headers can be viewed by clicking a "next page" link, which loads the next 40 headers.

The usage patterns characteristic of webmail applications provide an ideal setting for MapJAX-based data prefetching. Users desire low latency when loading new screens of message headers or viewing message bodies, but they generally have long "think times" while reading messages, providing a long interval during which the system can prefetch data. MapJAX permits an implementation of this application with little programmer effort. The application programmer simply accesses headers and bodies directly from MapJAX objects, without considering data transfer explicitly except for choosing a prefetching policy.

This application also illustrates the utility of MapJAX parallel **for** loops and locks. When loading a new screen of message headers, the programmer needs to append new header objects to the header list in the appropriate order. A non-MapJAX implementation needs to implement this ordering manually, which complicates program development by forcing the programmer to manually group object requests into messages and ensure that responses are processed in order.

By contrast, the MapJAX version of the code using a parallel **for** loop is correct, simple, and fast. The code in Figure 6 loads the first 40 headers into the header list in order, regardless of how many RPCs the MapJAX runtime chooses to issue to retrieve the headers. We assume that the header list is represented by a <DIV> element whose ID is "header_list_div". We show the corresponding Javascript code generated by the MapJAX compiler in Figure 7.

Figure 8 shows a version of the AJAX code which retrieves all headers using a single RPC. Already, this code can be seen to be more complicated than its Map-

```
var mailHeaders = new ArrayMap("mailHeaders",
                      new ReadAheadKPolicy(10),
                      "AJAXTransport");
var headerList =
   document.getElementByID("header_list_div");
var hdrLock = new RLULock();
pfor(var i = 0; i < 40; ++i) {
   hdrLock.reserve();
   var header = mailHeaders.access#(i);
   // Omit code to check for error condition
   hdrLock.lock#();
   var hdrDiv = document.createElement("div");
   // Omit code to initialize hdrDiv from header
   headerList.appendChild(headerDiv);
   hdrLock.unlock();
}
```

Figure 6: MapJAX implementation for loading a page of email headers.

JAX counterpart; adding facilities for tracking and ordering multiple requests (which might boost performance) would only make the situation worse.

5.4 Non-MapJAX Implementations

We close with a word on our non-MapJAX implementations of these applications. While the MapJAX applications use the automatic prefetching and caching features of the framework, we do not implement manual caching or prefetching in the non-MapJAX applications. Our rationale is that a manually-tuned application should always be able to perform as well as MapJAX, given a sufficient time investment: MapJAX and human programmers both have the same set of primitives available to them. We thus show the improvement possible given a programmer unwilling or unable to invest extensive time and energy in optimization.

Note that the absence of prefetching in the non-MapJAX examples greatly reduces their implementation complexity. Had we included this functionality, the benefits of of MapJAX would have been even more apparent. Caching on its own, by contrast, would be of little use on the test workloads presented in Section 6, as they do not reuse data, and the non-MapJAX applications do not suffer from its absence.

Caching and prefetching aside, we have attempted to write implementations that, while straightforward, avoid obvious performance pitfalls. We describe each in more detail below.

The non-MapJAX version of the webmail application takes the one-RPC approach to fetching message headers as discussed in Section 5.3 and illustrated in Figure 8. Message bodies are retrieved using one RPC per body, as multiple bodies are never fetched simultaneously.

The non-MapJAX version of the suggest application is almost identical to the MapJAX version, except that it explicitly sends an RPC to fetch completion requests.

```
var mailHeaders = new ArrayMap("mailHeaders",
                      new ReadAheadKPolicy(10),
                      "AJAXTransport");
var headerList =
   document.getElementByID("header_list_div");
var hdrLock = new RLULock();
for(var i = 0; i < 40; ++i) {
  hdrLock.reserve();
  _cx_thread_create(function() {
    mailHeaders.access(i, _cx_cont1);
  });
}

function _cx_cont1(header) {
    var contobj = new Object();
    contobj.header = header;
    hdrLock.lock(_cx_cont2, contobj);
}

function _cx_cont2(contobj) {
    var header = contobj.header;
    var hdrDiv = document.createElement(''div'');
    // Omit code to initialize hdrDiv from header
    headerList.appendChild(headerDiv);
    hdrLock.unlock();
}
```

Figure 7: Javascript code produced by the MapJAX compiler for loading a page of email headers.

In order to cope with network reordering, it maintains a version number on the completion display field that it increments each time it sends an RPC. The callback function for each RPC has a copy of the version number with which it is associated; when it is run, it checks the version number on the completion display and only updates the display if the version matches. For compatibility with the Google version of the application, whose data we use, RPC results contain Javascript code which is `eval'd` to update the display.

The non-MapJAX version of the mapping application is also close in implementation to the MapJAX version. Using much the same event-handler code, it moves and updates a collection of image objects on screen, the primary difference being that new image data are loaded by setting the "src" attribute of these objects to the appropriate URL, rather than using a MapJAX grid-map. While this implementation does not prefetch, the browser will cache images, and it also benefits from the request-canceling functionality that browser image objects support.

6 Experimental Results

In this section, we provide performance results that demonstrate the advantages of MapJAX. We test the applications described above under realistic Internet latency and bandwidth constraints and show that the unoptimized MapJAX versions perform comparably to the

```
var req = new XMLHttpRequest();
req.open("GET", "/mail-headers.cgi?idxs=0-40");
req.onreadystatechange = headerRPCHandler;
req.send(null);

function headerRPCHandler(req) {
  if (req.readyState == 4 && req.status == 200) {
    var jsonEncodedHeaders = req.responseText;
    var headers = decodeJSON(jsonEncodedHeaders);
    var headerList =
      document.getElementById("header_list_div");
    for (var i = 0; i < headers.length; i++) {
      var hdrDiv = document.createElement("div");
      //Skip code to init hdrDiv from headers[i]
      headerList.appendChild(hdrDiv);
    }
  } else {
    if (req.readyState == 4) {
      // handle RPC error condition
    }
  }
}
```

Figure 8: Non-MapJAX, single-RPC implementation for loading a page of email headers.

standard AJAX versions, while MapJAX performance optimizations can dramatically improve performance, by up to a factor of 2 in some cases. Additionally, we provide microbenchmarks demonstrating the utility of specific features of MapJAX.

These results illustrate two points. First, they show that the more intuitive programming model of MapJAX is provided with little overhead. Second, the results demonstrate that prefetching and other MapJAX optimizations are useful tools for these kinds of applications, and that substantial performance increases can be achieved with relatively simple-minded prefetching policies. Were one willing to expend additional effort devising more clever prefetching policies or tweaking other portions of the application, one might well achieve better performance than seen here. We do not consider that fact to detract from our results.

Our experiments were conducted using two PCs with Intel Pentium 4 3.8GHz CPUs and 4GB of RAM, running Fedora Core 4 with Linux kernel version 2.6.14-1.1656.FC4_smp. The server ran Apache Tomcat 5.5.17 with the tcnative extensions, and the client web browser was Firefox 2.0.0.1. To allow effective prefetching of images, and as recommended for good AJAX performance, we modified the Firefox configuration variables as indicated in Table 1, although we note that these changes are not mandatory. Before executing each experiment, we executed and discarded a complete run to warm up the server's cache.

To introduce network delays and bandwidth constraints, we used a 600 MHz Intel Pentium III running FreeBSD 4.11 and the dummynet [12] network emulator. This machine was connected to both the client and server by 100Mbit switched Ethernet.

Parameter	Value
network.http.pipelining	true
network.http.pipelining.maxrequests	16
network.http.max-connections	48
network.http.max-connections-per-server	48
network.http.max-persistent-c'xns-per-server	8

Table 1: Firefox configuration variables changed from defaults during testing.

6.1 Application Tests

6.1.1 Search Auto-Complete

As stated earlier, the search term auto-complete application is interesting because it represents an application with low bandwidth requirements but stringent latency requirements: search term completions that arrive after the user has input additional characters are of no use.

To evaluate the usefulness of MapJAX in this context, we measured the average latency of completion retrieval (the *average request latency*) of both a MapJAX and standard AJAX implementation under a range of latency and bandwidth parameters. Specifically, we tested using bandwidth values ranging from 256Kb/s to 1024Kb/s, which are typical of home broadband connections, and latencies of both 20 and 70 ms, which correspond to close and average-distance servers. The standard AJAX version of the application made no attempt to prefetch, and we tested the MapJAX version with both prefetching enabled (using the English language policy described above) and prefetching disabled.

To measure average request latency, we used a workload generated by typing 65 search terms from the April/May 2006 Google Zeitgeist list of popular search terms into the AJAX version of the application, resulting in a trace of 423 completion requests. The completions returned were those that would have been returned by Google's version of the application (they were retrieved from Google and cached at our server ahead of time). The average number of suggestions per suggestion set was 5.71, and the average suggestion set size was 264 bytes. We discarded the first 10 observed latencies to avoid measuring noise due to the application loading.

The results of this test are shown in Figure 9. First, we note that the non-MapJAX (i.e., standard AJAX) implementation average latencies and the MapJAX (no prefetching) average latencies are always within 3 ms of each other, indicating that MapJAX imposes minimal additional overhead in providing its programming model (even with the current unoptimized implementation). Second, we observe that when sufficient band-

width is available, prefetching significantly decreases the average latency (by close to half in the 1024Kb/s case). As expected, when sufficient bandwidth is not available, prefetching delays the servicing of actual cache misses and hurts performance. Future work will include automatic network performance measurements to allow programmers to scale back or disable prefetching in these cases. Additional results (not shown) show negligible CPU or memory overhead for the MapJAX implementation relative to the non-MapJAX implementation.

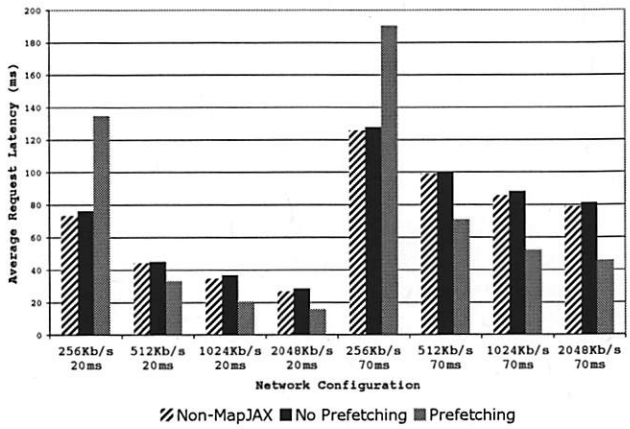


Figure 9: Average request latencies as generated by the auto-complete application on a sample workload under a range of simulated network conditions.

6.1.2 Mapping Application

In contrast to the auto-complete application, the mapping application represents a case in which the application is somewhat tolerant to latency (map tiles simply need to arrive before the user can scroll them completely off the screen) but has high bandwidth requirements.

To evaluate the utility of MapJAX in this context, we measured the average latency of map tile retrieval of both standard and MapJAX implementations under a range of bandwidth parameters (ranging from 256Kb/s to 8,192Kb/s) and a fixed latency of 70 ms. As before, the standard implementation made no attempt to prefetch, and we tested the MapJAX version with both prefetching enabled (using the OMNI policy described earlier, at various levels of prefetching) and prefetching disabled.

To measure the average request latency, we used a pair of user-generated workloads. Specifically, we asked a number of subjects to perform a simple navigation task using the mapping application with no bandwidth or delay constraints: scrolling from the MIT campus in Cambridge to the intersection of I-93 and MA-24 south of the city. The trace was formed by recording the GUI events (clicks and drags) thus generated. From this collection of traces, we chose two for testing. The first, which we call "hard," was generated by a user familiar with the area who was able to navigate at high speed. The second, which we call "easy," was generated by a user new to the area who navigated more slowly. The "hard" workload consisted of 467 GUI events resulting in 198 calls to access, and the "easy" workload consisted of 890 GUI events resulting in 196 calls to access. The average size of an image tile used by these workloads was 4,751 bytes.

Testing the application consisted of replaying these two traces and recording the access latency observed on non-canceled image tiles. Specifically, we used a viewport 8 tiles wide by 4 tiles high. The initial 32 images were loaded with prefetching disabled, and we did not record these latencies. We then enabled prefetching and replayed the trace. When computing average access latencies, we discarded the first 15 latencies generated by the trace to avoid measuring startup effects. The map tile images were those used by Google; they were downloaded and cached at our server ahead of time.

The results of these tests are presented in Figure 10 ("easy" trace) and Figure 11 ("hard" trace). Please note that the y-axis is log-scaled. To compensate for observed run-to-run variability, we report the average over three runs for each value, with error bars showing plus or minus one standard deviation.

We observe several interesting features of the graphs. First, at 256Kb/s, both workloads exhibit extremely high latencies with all implementations of the application, indicating that the 256Kb/s is insufficient bandwidth to support the workloads. Second, on the easy workload, MapJAX with prefetching disabled exhibits average latencies within 3% of the standard implementation except at 256Kb/s, where the average is highly variable and within 10%. On the hard workload, MapJAX with prefetching disabled exhibits average latencies within 12% of the standard implementation, except at 8192Kb/s, where it is within 17%.

These results indicate that the benefits of MapJAX are available with little overhead. Moreover, we believe that most of the overhead seen here is due to our unoptimized implementation and can be removed. Additional results (not shown) show that there is little to no startup overhead imposed by MapJAX, assuming that prefetching is disabled during startup. We also found MapJAX to impose negligible CPU or memory overheads relative to the non-MapJAX implementation.

As in the auto-complete application, as spare bandwidth becomes available, prefetching is able to dramatically reduce the average access latency. The "hard" workload sees a 62.5% reduction with Omni-2 at 2048Kb/s and an 83.8% reduction in latency with Omni-2 at 4096Kb/s. The "easy" workload sees a 21.2% re-

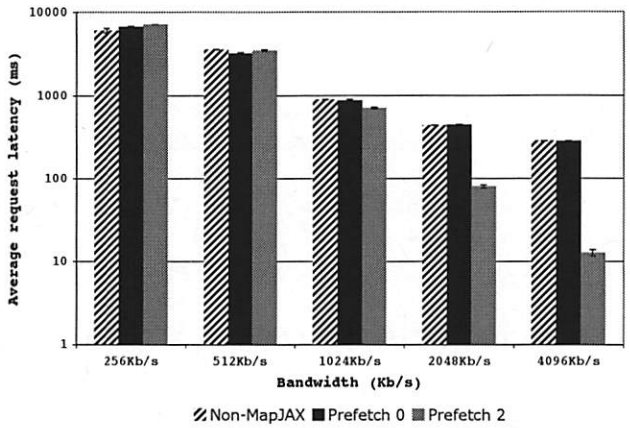


Figure 10: Results of running the mapping application with various prefetching policies on the "easy" workload using a simulated network with 70 ms latency and varied bandwidth constraints.

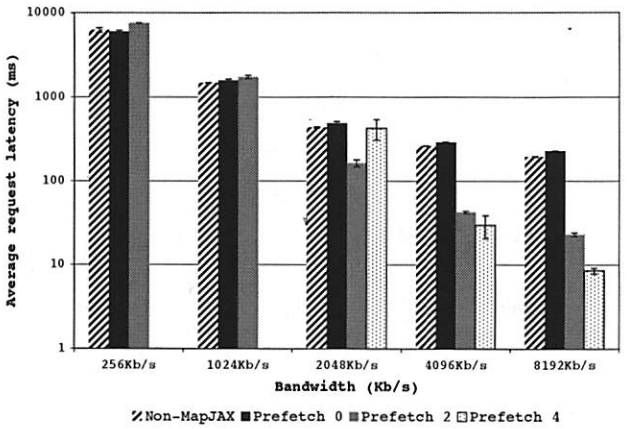


Figure 11: Results of running the mapping application with various prefetching policies on the "hard" workload using a simulated network with 70 ms latency and varied bandwidth constraints. The "prefetch 4" result is not shown for the 256Kb/s and 1024Kb/s cases as performance was extremely poor.

duction with Omni-2 at 1024Kb/s, an 81.7% reduction with Omni-2 at 2048Kb/s, and a 95.5% reduction in latency with Omni-2 at 4096Kb/s. By contrast, prefetching increases access latency when spare bandwidth is not available, as would be expected; again, future work will allow programmers to scale back or disable prefetching in this case.

6.1.3 Mail

We exercise some features of the mail application in the microbenchmarks, below. Full-application tests provided no additional information beyond that obtained from the mapping and auto-complete applications and are omitted here.

6.2 Microbenchmarks

6.2.1 Parallel *For* loops and Request Combining

To evaluate parallel **for** loops and request combining, we measured the total time required to load a page of 40 message headers in the MapJAX implementation of our email application with both parallel and non-parallel **for** loops. In the parallel case, we tested with request combining both enabled and disabled. (Non-parallel **for** loops provide no opportunity for request combining, so we did not test that combination.) To eliminate network effects, we introduced no latency or bandwidth constraints and disabled prefetching.

The results of this experiment are shown in Table 2. Parallel **for** loops and request combining both provide a clear advantage: parallel **for** loops provide an order of magnitude speedup over non-parallel loops, as they are able to fetch all items required by the loop immediately, rather than waiting for one RPC to complete before sending the next. Request combining provides a speedup greater than a factor of two, as it cuts the number of RPCs issued from 40 to one. Had we used simulated network delays, the advantage of MapJAX would have been even greater.

For comparison, the non-MapJAX version of the code, which sends a single RPC for all 40 headers, averaged 77.6 ms over 10 trials with a standard deviation of 0.52. We believe most of the difference between that value and MapJAX parallel **for** loops with request combining is due to implementation artifacts and can be eliminated.

	Combining	**No Combining**
pfor	117.8 ± 5.51	277.2 ± 12.18
for	*(Not Run)*	1465.0 ± 18.90

Table 2: Effectiveness of *pfor* loops and request combining. Values shown are the times to load a page of 40 message headers in our email application, averaged over 10 trials and given in milliseconds $\pm$ one standard deviation.

6.2.2 Locks

To evaluate the overhead of our lock implementation, we compared the performance of the MapJAX version of the mapping application on the "hard" trace against the performance of a version that manually ordered updates instead of using locks. To eliminate network effects, we added no bandwidth or delay constraints and disabled prefetching. Averaging over three runs, the version using locks had an average access latency of 59.76 ms, while

the version using manual ordering had an average access latency of 58.20 ms. Standard deviations were 1.82 and 1.80, respectively. We conclude that our lock implementation adds negligible overhead.

6.2.3 Request Canceling

To evaluate the contribution made by request canceling, we modified the MapJAX version of our mapping application to not cancel requests and ran the "easy" workload on a simulated 256Kb/s, 70ms network with prefetching disabled. (This simulated network provides insufficient bandwidth to support the trace, and thus request canceling is particularly important). The average request latency (averaged over three runs) was 33,397.92 ms, which is far greater than the 6605.40 ms obtained with canceling enabled. Additional experiments (not shown) found the importance of canceling to decrease as bandwidth increased, which is the expected result: in an environment where bandwidth is plentiful, there is no need to conserve it. We conclude that request canceling brings performance benefits worthy of the additional complexity it adds to the model.

7 Related Work

The trade-off between threading and event-based models has been well studied, recently in [14], which considered the issue in a server context. The MapJAX compiler is similar in some respects to TAME [7], which carries out a similar continuation-elimination function for code written using the libasync [8] C++ library for event-driven network servers.

To our surprise, we were unable to find previous work on locks with RLU semantics. In [6], the authors propose a lock reservation scheme to decrease the overhead of lock acquisition in Java VMs, their scheme only allows one thread to hold a lock reservation, whereas in our scheme multiple threads can reserve a lock. Our locks might appear similar to callback-based, asynchronously-acquired locks (e.g. as in [10]) but in fact they provide stronger semantics. Specifically, we guarantee that statements between the calls to reserve and lock will execute strictly before any statements after the call to lock. By contrast, an asynchronously-acquired lock makes no such guarantee about when its callback will be executed.

There have been several proposals of programming models for writing rich web applications, from ambitious efforts that provide fresh display layout and programming languages to smaller, lighter-weight efforts that try to smooth the rough edges of the current model. To our knowledge no system exists that provides the shared data abstraction, elimination of callbacks, and array of performance optimizations available in MapJAX.

Examples of ambitious web programming systems include Java applets and Adobe Flash. Such systems have the advantage of starting from a clean slate, which allows them to ignore the imperfections of the standard web development model. However, on the Internet, incremental deployability is often key: technologies that require users to install new software and developers to learn new languages often do not succeed. Additionally, systems of this type can be difficult to integrate cleanly into HTML-based pages, which renders them unattractive from a designer's perspective. Flash comes closest to the MapJAX programming model of any of the current systems: it provides data structure objects that can be bound to server-side data. Flash lacks MapJAX's support for prefetching, parallel **for**, and RLU locks, and it does not eliminate callbacks. Additionally, it requires a separate browser plugin to run and requires the programmer to learn an additional language.

At the other end of the spectrum, the growth in popularity of AJAX has given rise to numerous small libraries that attempt to put a friendlier face on AJAX development, including Prototype [13], Mochikit [10], JSON-RPC-Java [9], and Direct Web Remoting [15]. In general, these libraries tend to provide some subset of three classes of features.

First, some offer a set of reusable user interface controls for AJAX applications, such as a table that is dynamically filled in with data from the server. While such controls are similar in spirit to MapJAX shared data structures, they are hardly a full-fledged programming model. Second, some libraries attempt to resolve some of the deficiencies in the Javascript programming language; e.g., they might add extra functions for handling strings, accessing HTML elements, managing asynchronous tasks, or logging. Finally, some libraries include support for some variant of RPC built on top of AJAX requests. The level of RPC abstraction provided is widely variable: the Prototype and Mochikit frameworks free the programmer from some of the event-handling associated with managing AJAX requests but keep the asynchronous HTTP-request model, whereas DWR and JSON-RPC-Java both extend Java RMI [16] to the browser. In all cases, however, these libraries at best provide a more pleasant interface over what is essentially an asynchronous RPC call, along with problems of that abstraction.

In between the two above extremes are web development platforms such as the Google Web Toolkit [3], Ruby on Rails (RoR) [4] and OpenLaszlo [11]. These systems use existing browser technologies to deploy their applications, but they provide a higher level of abstraction than the small libraries discussed above. The Google Web Toolkit provides a Java-to-AJAX compiler, but it does not include MapJAX-style shared data structures,

and it provides callback-based RPCs. Ruby on Rails is a rapid development framework for database-backed applications and thus includes some aspects of a data model, but it supports neither prefetching nor callback elimination. Finally, OpenLaszlo provides a compiler from their own language to AJAX, although again without shared data structures, and it retains asynchronous callbacks.

8 Conclusion

This paper has presented MapJAX, a new programming environment for AJAX-style web applications. In contrast to current systems based on asynchronous RPCs, MapJAX provides application programmers with the abstraction of logical data structures shared between client and server, accessed through a familiar programming model based on objects, threads, and locks. MapJAX also includes a number of additional features (parallel **for** loops, data prefetching, and request canceling) that help programmers implement highly-responsive applications.

There are several areas of MapJAX that warrant future work. First, we would like the system to better adapt to changing network conditions: the runtime should characterize the performance of the network and adapt prefetching accordingly, either automatically or by exposing this information to the application. Second, the cache could be extended to persist on disk across reloads of the application. Finally, we would like to extend MapJAX to handle writable data structures.

In summary, we have implemented three prototypical AJAX applications using both standard AJAX techniques and MapJAX. We tested them under realistic Internet latency and bandwidth constraints and found that the unoptimized MapJAX versions of the applications performed comparably to the standard AJAX versions, and that MapJAX performance optimizations could dramatically improve performance, by up to a factor of 2 in some cases. Finally, we have performed microbenchmarks exercising each of the performance optimizations we provide and have shown the contributions made by each. We believe our results show that MapJAX meets its goals of reducing the development complexity while simultaneously improving the performance of AJAX web applications.

9 Acknowledgments

The authors thank Robert Morris and Emil Sit for valuable conversations during the development of this system, as well as the anonymous reviewers for their comments.

References

[1] FESTA, P. Will AJAX help google clean up? http://news.com.com/Will+AJAX+help+Google+clean+up/2100-1032_3-5621010.html, March 2005.

[2] GARRETT, J. Ajax: a new approach to web applications, February 2005. http://www.adaptivepath.com/publications/essays/archives/000385.php.

[3] GOOGLE, INC. Google web toolkit. http://code.google.com/webtoolkit.

[4] HASSON, D. Ruby on Rails. http://rubyonrails.org.

[5] JSON. http://www.json.org.

[6] KAWACHIYA, K., KOSEKI, A., AND ONODERA, T. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), ACM Press, pp. 130–141.

[7] KROHN, M., KOHLER, E., AND KAASHOEK, M. F. Simplified event programming for busy network applications. In *Proceedings of the 2007 USENIX Annual Technical Conference* (Santa Clara, CA, USA, June 2007), USENIX.

[8] MAZIERES, D. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Annual Technical Conference* (Boston, MA, June 2001), USENIX.

[9] METAPARADIGM PTE LTD. JSON-RPC-Java. http://oss.metaparadigm.com/jsonrpc/.

[10] Mochi Media, LLC. http://mochikit.com.

[11] Open Laszlo. http://openlaszlo.org.

[12] RIZZO, L. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review 27*, 1 (1997), 31–41.

[13] STEPHENSON, S. Prototype JavaScript Library. http://prototype.conio.net.

[14] VON BEHREN, R., CONDIT, J., AND BREWER, E. Why events are a bad idea for high-concurrency servers, 2003.

[15] WALKER, J., AND GOODWIN, M. DWR - Easy AJAX for Java. http://getahead.ltd.uk/dwr.

[16] WOLLRATH, A., RIGGS, R., AND WALDO, J. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems (COOTS)* (1996), USENIX Association, pp. 219–232.

[17] ZEUS TECHNOLOGY LIMITED. Zeus Web Server. http://www.zeus.co.uk.

Sprockets: Safe extensions for distributed file systems

Daniel Peek[‡], *Edmund B. Nightingale*[‡], *Brett D. Higgins*[‡], *Puspesh Kumar*[†], *and Jason Flinn*[‡]
University of Michigan[‡] *and IIT Kharagpur*[†]

Abstract

Sprockets are a lightweight method for extending the functionality of distributed file systems. They specifically target file systems implemented at user level and small extensions that can be expressed with up to several hundred lines of code. Each sprocket is akin to a procedure call that runs inside a transaction that is always rolled back on completion, even if sprocket execution succeeds. Sprockets therefore make no persistent changes to file system state; instead, they communicate their result back to the core file system through a restricted format using a shared memory buffer. The file system validates the result and makes any necessary changes if the validations pass. Sprockets use binary instrumentation to ensure that a sprocket can safely execute file system code without making changes to persistent state. We have implemented sprockets that perform type-specific handling within file systems such as querying application metadata, application-specific conflict resolution, and handling custom devices such as digital cameras. Our evaluation shows that sprockets can be up to an order of magnitude faster to execute than extensions that utilize operating system services such as `fork`. We also show that sprockets allow fine-grained isolation and, thus, can catch some bugs that a `fork`-based implementation cannot.

1 Introduction

In recent years, the file systems research community has proposed a number of new innovations that extend the functionality of storage systems. Yet, most production file systems have been slow to adopt such advances. This slow rate of change is a reasonable precaution because the storage system is entrusted with the persistent data of a computer system. However, if file systems are to adapt to new challenges presented by scale, widespread storage of multimedia data, new clients such as consumer electronic devices, and the need to efficiently search through large data repositories, they must change faster.

In this paper, we explore a method called *sprockets* that safely extends the functionality of distributed file systems. Our goal is to develop methodologies that let third-party developers create binaries that can be linked into the file system. Sprockets target finer-grained extensions than those supported by Watchdogs [3] and user level file system toolkits [6, 17], which offer extensibility at the granularity of VFS calls such as `read` and `open`. Sprockets are intended for smaller, type-specific tweaks to file system behavior such as querying application-specific metadata and resolving conflicts in distributed file systems. Sprockets are akin to procedure calls linked into the code base of existing file systems, except that they *safely* extend file system behavior.

While one might think that extending the behavior of a distributed file system requires one to alter kernel functionality, many distributed file systems such as AFS [10], BlueFS [20], and Coda [13] implement their core functionality at user level. It is these file systems that we target; extending file system functionality in the kernel can be accomplished through other methods [2, 5, 22, 25]. In many ways, extending user level code is easier than extending kernel code since the extension implementation can use operating system services to sandbox extensions to user level components. However, we have found that existing services such as `fork` are often prohibitively expensive for commonly-used file system extensions that are only a few hundred lines of code. Further, isolation primitives such as `chroot` can be insufficiently expressive to capture the range of policies necessary to support some file system extensions.

The sprocket extension model is based upon software-fault isolation. Sprockets are easy to implement since they are executed in the address space of the file system. They may query existing data structures in the file system and reuse powerful functions in the code base that manipulate the file system abstractions. To ensure safety, sprockets execute inside a transaction that is always partially rolled back on completion, even if an extension executes correctly. A sprocket may execute arbitrary user level code to compute its results, but it must express those results in a limited buffer shared with the file system. Only the shared buffer is not rolled back at the end of sprocket execution. The results are verified by the core file system before changes are made to file state.

We have used sprockets to implement three ideas from the file systems research community: transducers [7], application-specific resolvers [14], and automatic translation of file system operations to device-specific protocols. Our performance results show that sprockets are up to an order of magnitude faster than safe execution using operating system services such as `fork`, yet they can enforce stricter isolation policies and prevent some bugs that `fork` does not.

2 Design goals

What is the best way to extend file system functionality? To answer this question, we first outlined the goals that we wished to achieve in the design of sprockets.

2.1 Safety

Our most important goal is safe execution of potentially unreliable code. The file system is critical to the reliability of a computer system — it should be a safe repository to which persistent data can be entrusted. A crash of the file system may render the entire computer system unusable. A subtle bug in the file system can lead to loss or corruption of the data that it stores [27]. Since the file system often stores the only persistent copy of data, such errors are to be avoided at all costs.

We envision that many sprockets will be written by third-party developers who may be less familiar with the invariants and details of the file system than core developers. Sprockets may also be executed more rarely than code in the core file system, meaning that sprocket bugs may go undetected longer. Thus, we expect the incidence of bugs in sprockets to be higher than that in the core file system. It is therefore important to support strong isolation for sprocket code. In particular, a programming error in a sprocket should never crash the file system nor corrupt the data that the file system stores. A buggy sprocket may induce an incorrect change to a file on which it operates since the core file system cannot verify application-specific semantics within a file. However, the core file system can verify that any changes are semantically correct given its general view of file system data (e.g., that a file and its attributes are still internally consistent) and that the sprocket only modifies files on which it is entitled to operate.

Like previous systems such as Nooks [25], our design goal is to protect against buggy extensions rather than those that are overtly malicious. In particular, our design makes it extremely unlikely, but not impossible, for a sprocket to compromise the core file system. Our design also cannot protect against sprockets that intentionally leak unauthorized data through covert channels.

2.2 Ease of implementation

We also designed sprockets to minimize the cost of implementation. We wanted to make only minimal changes to the existing code of the core file system in order to support sprockets. We eliminated from consideration any design that required a substantial refactoring of file system code or that added a substantial amount of new complexity. We also wanted to minimize the amount of code required to write a new sprocket. In particular, we decided to make sprocket invocation as similar to a procedure call as possible.

Sprockets can call any function implemented as part of the core file system. Distributed file systems often consist of multiple layers of data structures and abstractions. A sprocket can save substantial work if it can reuse high-level functions in the core file system that manipulate those abstractions.

We also let sprockets access the memory image of the file system that they extend in order to reduce the cost of designing sprocket interfaces. If a sprocket could only access data passed to it when it is called, then the file system designer must carefully consider all possible future extensions when designing an interface in order to make sure that the set of data passed to the sprocket is sufficient. In contrast, by letting sprockets access data not directly passed to them, we enable the creation of sprockets that were not explicitly envisioned when their interfaces were designed.

2.3 Performance

Finally, we designed sprockets to have minimal performance impact on the file system. Most of the sprockets that we have implemented so far can be executed many times during simple file system operations. Thus, it is critical that the time to execute each sprocket be small so as to minimize the impact on overall file system performance. Fortunately, most of the sprockets that we envision can be implemented with only a few hundred lines of code or less. These features led us to bias our choice of designs toward one that had a low constant performance cost per sprocket invoked, but a potentially higher cost per line of code executed.

An alternative to the above design bias would be batch processing so that each sprocket does much more work when it is invoked. Batching reduces the need to minimize the constant performance cost of executing a sprocket by amortizing more work across the execution of a single sprocket. However, batching would considerably increase implementation complexity by requiring us to refactor file system code wherever sprockets are used.

3 Alternative Designs

In this section, we discuss alternative designs that we considered, and how these led to our current design.

3.1 Direct procedure call

There are many possible implementations for file system extensions. The most straightforward one is to simply link extension code into the file system and execute the extension as a procedure call. This approach is similar to how operating systems load and execute device drivers. Direct execution as a procedure call minimizes the cost of implementation and leads to good performance. However, this design provides no isolation: a buggy extension can crash the file system or corrupt data. As safety is our most important design goal, we considered this option no further.

3.2 Address space sandboxing

A second approach we considered is to run each extension in a separate address space. A simple implementation of this approach would be to `fork` a new process and call `exec` to replace the address space with a pristine copy for extension execution. This type of sandboxing is used by Web servers such as Apache to isolate untrusted CGI scripts. A more sophisticated approach to address sandboxing can provide better performance. In the spirit of Apache FastCGI scripts, the same forked process can be reused for several extension executions.

However, both forms of address space sandboxing suffer from two substantial drawbacks. First, they provide only minimal protection from persistent changes made by an extension through the execution of a system call. In particular, a buggy extension could corrupt file system data by incorrectly overwriting the data stored on disk. Potentially, such modifications could even violate file system invariants and lead to a crash of the file system when it reads the corrupted data. While operating systems do provide some tools such as the `chroot` system call and changing the effective userid of a process, these tools have a coarse granularity. It is hard to allow an extension access to only some operations, but not others. For instance, one might want to allow an extension that does transcoding to access only an input file in read mode and an output file in write mode. Restricting its privilege in this manner using the existing API of an operating system such as Linux requires much effort. Thus, address space sandboxing does not provide completely satisfactory isolation on current operating systems.

A second drawback of address space sandboxing is that it considerably increases the difficulty of extension implementation. If the extension and the file system exist in separate address spaces, then the extension cannot access the file system's data structures, meaning that all data it needs for execution must be passed to it when it starts. Further, the extension cannot reuse functions implemented as part of the file system. While one could place code of potential interest to extensions in a shared library, the implementation cost of such a refactoring would be large.

3.3 Checkpoint and rollback

The above drawback led us to refine our design further to allow extensions to execute in the address space of the original file system. As before, the file system forks a new process to run the extension. However, instead of calling `exec` to load the executable image of the extension, the extension is instead dynamically loaded into the child's address space and directly called as a procedure. After the extension finishes, the child process terminates.

One way to view this implementation is that each extension executes as a transaction. However, in contrast to transactions that typically commit on success, these transactions are *always* rolled back. Since `fork` creates a new copy-on-write image, any modifications made to the original address space by the extension are isolated to the child process — the file system code never sees these modifications.

Extensions may often make persistent changes to file system state. Since it is unsafe to allow the extension to make such changes directly, we divide extension execution into two phases. During the first phase, the extension generates a description of the changes to persistent file state that it would like to make. This description is expressed in a format specific to each extension type that can be interpreted by the core file system. In the second phase, the core file system reads the extension's output and validates that it represents an allowable modification. This validation is specific to the function expected of the extension and may be as simple as checking that changes are made only to specific files or that returned values fall within a permissible range.

If all validations pass, the core file system applies the changes to its persistent state. This approach is similar to that taken by an operating system during a system call. From the point of view of the operating system, the application making the call can execute arbitrary untrusted code; yet, the parameters of the system call can be validated and checked for consistency before any change to persistent state is made as a result of the call. This implementation relies on the fact that while the particular *policy* that determines what changes need to be made can be arbitrarily complex (and thus is best described with code), the *set of changes* that will be made as a result of that policy is often limited and can be expressed using a simple interface.

For example, consider the task of application-specific resolution, as is done in the Coda file system [14]. A

resolver might merge conflicting updates made to the same file by reading both versions, performing some application-specific logic, and finally making changes that merge the conflicting versions into a single resolved file. While the application logic behind the resolution is specific to the types of files being merged, the possible result of the resolution is limited. The conflicting versions of the file will be replaced by new data. Thus, a extension that implements application-specific resolution can express the changes it wishes to make in a limited format such as a patch file that is easily interpreted by generic file system code. That core file system then verifies and applies the patch.

In the transactional implementation, the extension needs some way to return its result so that it can be interpreted, validated, and applied by the file system. We allow this by making the rollback at the end of extension execution partial. Before the extension is executed, the parent process allocates a new region of memory that it shares with its child. This region is exempted from the rollback when the extension finishes. The parent process instead reads, verifies, and applies return values from this shared region, and then deallocates it.

The transactional implementation still has some drawbacks. Like address space isolation, we must rely on operating system sandboxing to limit the changes that an extension can make outside its own address space.

A second drawback occurs when the file system code being extended is multithreaded. The extension operates on a copy of the data that existed in its parent's address space at the time it was forked. However, this copy could potentially contain data structures that were concurrently being manipulated by threads other than the one that invoked the extension. In that case, the state of the data structures in the extension's copy of the address space may violate expected invariants, causing the extension to fail. Ideally, we would like to fork an extension only when all data structures are consistent. One way to accomplish this would be to ask extension developers to specify which locks need to be held during extension execution. We rejected this alternative because it requires each extension developer to correctly grasp the complex locking semantics of the core file system. Instead, the extension infrastructure performs this task on behalf of the developer by relying on the heuristic that threads that modify shared data should hold a lock that protects that data. We use a barrier to delay the `fork` of an extension until no other threads currently hold a lock. This policy is sufficient to generate a clean copy as long as all threads follow good programming practice and acquire a lock before modifying shared data.

A final substantial drawback is that `fork` is a heavyweight operation on most operating systems: when an extension consists of only a few hundred lines of code, the time to fork a process may be an order of magnitude greater than the time to actually execute the extension. During `fork`, the Linux operating system copies the page table of the parent process—this cost is roughly proportional to the size of the address space. For instance, we measured the time to fork a 64 MB process as 6.3 ms on a desktop running the Linux 2.4 operating system [19]. This cost does not include the time that is later spent servicing page faults due to flushing the TLB and implementing copy-on-write. Overall, while the transactional implementation offers reasonably good safety and excellent ease of implementation, it is not ideal for performance because of the large constant cost of `fork`.

4 Sprocket design and implementation

Performance considerations led to our current design for sprocket implementation, which is to use the transactional model described in the previous section but to implement those transactions using a form of software fault isolation [26] instead of using address space isolation through `fork`.

4.1 Adding instrumentation

We use the PIN [16] binary instrumentation tool to modify the file system binary. PIN generates new text as the program executes using rules defined in a PIN *tool* that runs in the address space of the modified process. The modified text resides in the process address space and executes using an alternate stack. The separation of the original and modified text allows PIN to be turned on and off during program execution. We use this functionality to instrument the file system binary only when a sprocket is executing. Instrumenting and generating new code for an application is a very expensive operation, but the instrumentation must be performed only once for each instruction. Unfortunately, since PIN is designed for dynamic optimization, it does not support an option (available in many other instrumentation tools) to statically pre-instrument binaries before they start running. To overcome this artifact of the PIN implementation, we can pre-instrument sprockets by running them once on dummy data when the file system binary is first loaded.

We have implemented our own PIN tool to provide a safe execution environment in which to run sprockets. When a sprocket is about to be executed, the PIN instrumentation is activated. Our PIN tool first saves the context of the calling thread (e.g., register states, program counter, heap size, etc.). As the sprocket executes, for each instruction that writes memory, our PIN tool saves the original value and the memory location that was modified to an undo log.

When the sprocket completes execution, each memory location in the undo log is restored to its original value and the program context is restored to the point before the sprocket was executed. The PIN tool saves the

```
/* Arguments passed to sprocket */
help_args.buf = NULL;
help_args.len = 0;
help_args.file1_size = server_attr.size;
help_args.file2_size = client_file_stat.st_size;

/* Set up return buffer and invoke sprocket */
SPROCKET_SET_RETURN_DATA (help_args.shared_page, getpagesize());
rc = DO_SPROCKET(resolver_helper, &help_args);

if (rc == SPROCKET_SUCCESS) {
    /* Verify and read return values */
    get_needed_data(help_args.shared_page, &help_args,
                    NULL, fid, &server_attr, path);
} else {
    /* handle sprocket error */
    ...
```

Figure 1. Example of sprocket interface

sprocket's return code and passes this back to the core file system as the return value of the sprocket execution. Like the fork implementation, the sprocket infrastructure allocates a special region of memory in the process address space for results — modifications to this region are not rolled back at the end of sprocket execution. If sprocket execution is aborted due to an exception, bug, or timeout, the PIN tool substitutes an error return code. Prior to returning, the PIN tool disables instrumentation so that the core file system code executes at native speed.

The ability to dynamically enable and disable instrumentation is especially important since sprockets often call core file system functions. When the sprocket executes, PIN uses a slow, instrumented version of the function that is used during all sprocket executions. When the function is called by the core file system, the original, native-speed implementation is used. Instrumented versions are cached between sprocket invocations so that the instrumentation cost need be paid only once.

Running the instrumented sprocket code, which saves modified memory values to an undo log, is an order of magnitude slower than running the native, uninstrumented version of the sprocket. However, since most sprockets are only a few hundred lines of code, the total slowdown due to instrumentation can be substantially less than the large, constant performance cost of `fork`.

We perform a few optimizations to improve the performance of binary instrumentation. We observed that many modifications to memory occur on the stack. By recording the location of the stack pointer when the sprocket is called, we can determine which region of the stack is unused at the point in time when the sprocket executes. We neither save nor restore memory in this unused region when it is modified by the sprocket. Similarly, we avoid saving and restoring areas of memory the sprocket allocates using `malloc`. Finally, we avoid duplicate backups of the same address.

Binary instrumentation also allows us to implement fine-grained sandboxing of sprocket code. Rather than rely on operating system facilities such as `chroot`, we use PIN to trap all system calls made by the sprocket. If the system call is not on a whitelist of allowed calls, described in Section 4.3, the sprocket is terminated with an error. Calls on the whitelist include those that do not change external state (e.g., `getpid`). We also allow system calls that enable sprockets to read files but not modify them.

4.2 Sprocket interface

Figure 1 shows an example of how sprockets are used. From the point of view of the core file system, sprocket invocation is designed to appear like a procedure call. Each sprocket is passed a pointer argument that can contain arbitrary data that is specific to the type of sprocket being invoked. Since sprockets share the file system address space, the data structure that is passed in may include pointers. Alternatively, a sprocket can read all necessary data from the file server's address space.

The `SPROCKET_SET_RETURN_DATA` macro allocates a memory region that will hold the return value. In the example in Figure 1, this region is one memory page in size. The `DO_SPROCKET` macro invokes the sprocket and rolls back all changes except for data modified in

the designated memory region. In the example code, the core file system function `get_needed_data` parses and verifies the data in the designated memory region, then deallocates the region. As shown in Figure 1, the core file system may also include error handling code to deal with the failure of sprocket execution.

4.3 Handling buggy sprockets

Sprockets employ a variety of methods to prevent erroneous extensions from affecting core file system behavior and data. Because changes to the process address space made by a sprocket are rolled back via the undo log, the effects of any sprocket bug that stomps on core file system data structures in memory will be undone during rollback. Similarly, a sprocket that leaks memory will not affect the core file system. Because the data structures used by malloc are kept in the process address space, any memory allocated by the sprocket is automatically freed when the undo log is replayed and the address space is restored. Additional pages acquired by memory allocation during sprocket execution are deallocated with the `brk` system call.

Other types of erroneous extensions are addressed by registering signal handlers before the execution of the socket. For instance, if a sprocket dereferences a NULL pointer or accesses an invalid address, the registered `segfault` handler will be called. This handler sets the return value of the sprocket to an error code and resets the process program counter in the saved execution context passed into the handler to the entry point of the rollback code. Thus, after the handler finishes, the sprocket automatically rolls back the changes to its address space, just as if the sprocket had returned with the specified error code. To handle sprockets that consume too much CPU (e.g., infinite loops), the sprocket infrastructure sets a timer before executing the extension.

The final type of errors currently handled by sprockets are erroneous system calls. Sprockets allow fine-grained, per-system-call capabilities via a *whitelist* that specifies the particular system calls that a sprocket is allowed to execute. We enforce the whitelist by using the PIN binary instrumentation tool to insert a check before the execution of each system call. If the system call being invoked by a sprocket is not on its whitelist, the sprocket is aborted and rolled back with an error code.

We support per-call handling for some system calls. For instance, we keep track of the file descriptors opened by each sprocket. If a sprocket attempts to close a descriptor that it has not itself opened, we roll back the sprocket and return an error. Similarly, after the sprocket finishes executing, our rollback code automatically closes any file descriptors that the sprocket has left open, preventing it from leaking a consumable resource.

One remaining way in which a buggy sprocket can affect the core file system code is to return invalid data via the shared memory buffer. Unfortunately, since the return values are specific to the type of sprocket being invoked, the sprocket execution code cannot automatically validate this buffer. Instead, the code that invokes the sprocket performs a sprocket-specific validation before using the returned values. For instance, one of our sprockets (described in Section 5.2) returns changes to a file in a `patch`-compatible file format. The code that was written to invoke that particular sprocket verifies that the data in the return buffer is, in fact, compatible with the patch format before using it.

4.4 Support for multithreaded applications

Binary instrumentation introduces a further complication for multithreaded programs: other threads should never be allowed to see modifications made by a sprocket. This is an important consideration for file systems since most clients and servers are designed to support a high level of concurrency. We first discuss our current solution to multithreaded support, which is most appropriate for uniprocessors, and then discuss how we can extend the sprocket design in the future to better support file system code running on multiprocessors.

Our current design for supporting multithreaded applications relies on the observation that the typical time to execute a sprocket (0.14–0.62 ms in our experiments) is much less than the scheduling quantum for a thread. Thus, if a thread would ordinarily be preempted while a sprocket is running, it is acceptable to let the thread continue using the processor for a small amount of time in order to complete the sprocket execution. If the sprocket takes too long to execute, its timer expires and the sprocket is aborted. Effectively, we extend our barrier implementation so that sprockets are treated as a critical section; no other thread is scheduled until the sprocket is finished or aborted. Although our barrier implementation is slightly inefficient due to locking overheads, we would require a more expressive interface such as Anderson's scheduler activations [1] to utilize a kernel-level scheduling solution.

On a multiprocessor, the critical section implementation has the problem that all other processors must idle (or execute other applications) while a sprocket is run on one processor. If sprockets comprise a small percentage of total execution time, this may be acceptable. However, we see two possible solutions that would make sprockets more efficient on multiprocessors. One possibility would be to also instrument core file system code used by other threads during sprocket execution. If one thread reads a value modified by another, the original value from the undo log is supplied instead. This solution allows other threads to make progress during sprocket execution, but imposes a performance penalty on those threads since

they also must be instrumented while a sprocket executes.

An alternative solution is to have sprockets modify data in a shadow memory space. Instructions that read modified values would be changed to read the values from the shadow memory rather than from the normal locations in the process address space. For example, Chang and Gibson [4] describe one such implementation that they used to support speculative execution.

5 Sprocket uses

In order to examine the utility of sprockets, we have taken three extensions proposed by the file systems research community and implemented them as sprockets. The next three subsections describe our implementation of transducers, application-specific conflict resolution, and device-specific protocols using sprockets.

For these examples, we chose to extend the Blue distributed file system [20] because we are familiar with its source code and, like many distributed file systems, it performs most functionality at user level. Further, its focus on multimedia and consumer electronic clients [21] is a good opportunity to explore the use of sprockets to support the type-specific functionality for personal multimedia content.

5.1 Transducers

The first type of sprocket implements application-specific semantic queries over file system data. The functionality of this sprocket is similar to that of a transducer in the Semantic File System [7] or in Apple's Spotlight [24] in that it allows users to search and index type-specific attributes contained within files. For example, one might wish to search for music produced by a particular artist or photos taken on a specific date. This information is stored as metadata within each file (in the ID3 tag of music files and in the JPEG header of photos). However, since the organization of metadata is type-specific, the file system must understand the metadata format before it can search or index files of a given type. Our sprocket transducers extend BlueFS by providing this type-specific knowledge.

We have implemented our transducer sprocket as an extension to the BlueFS persistent query facility [21]. Persistent queries notify applications about modifications to data stored within the file system. An application running on any client that is interested in receiving such notifications specifies a semantic query (e.g., all files that end in ".mp3") and the set of events in which it is interested (e.g., file existence and new file creation). The query is created as a new object within the file system. The BlueFS server evaluates the query and adds log records for all matching events. For instance, in the above example, the server would initially add a log record to the query for every MP3 file accessible to the user who created the query, and then incrementally add a new record every time a new MP3 file is created. As in the above example, a query can be used either statically (to evaluate the current state of the file systems) and/or dynamically (to receive notifications when modifications are made to the file system).

In the existing BlueFS implementation, a persistent query could only be specified as a semantic query over file system metadata such as the file name and owner. Such file metadata is generic, meaning that the file server can easily interpret the metadata for all files it stores. A generic routine in the server is called to evaluate the query each time there is a potential match; the routine returns true if the file metadata matches the semantic query specified and false otherwise. However, this generic approach cannot easily be used for type-specific metadata such as the ID3 tags in music files, because the format of tags is opaque to the file server.

To support type-specific metadata, we extended the persistent query interface to allow applications to optionally specify a sprocket that will be called to help evaluate the query. For each potential match, the server first performs the generic type-independent evaluation described above (for instance, the query might verify that the filename ends in ".mp3"). If the generic evaluation returns true, the server invokes the sprocket specified for the query.

The query sprocket reads the type-specific metadata from the file, evaluates the contents, and returns a boolean value that specifies whether or not the file matches the query. If the sprocket returns true, the server appends a record to the persistent query object; the server takes no action if the sprocket returns false.

Reading data from a server file is a relatively complex operation. File data may reside in one of three places: in a file on disk named by the unique BlueFS identifier for that file, in the write-ahead log on the server's disk, or in a memory cache that is used to improve read performance. Executing the sprocket within the address space of the server improves performance because the sprocket can reuse the server's memory cache to avoid reading data from disk. Further, when the cache or write-ahead log contains more recent data than on disk, executing the sprocket in the server's address space avoids the need to flush cached data and truncate the write-ahead log. If the sprocket were a stand-alone process that only read data from the on-disk file, then it would read stale data if the cache were not flushed and the write-ahead log truncated.

The sprocket design considerably reduces the complexity of transducers in BlueFS. The sprocket can reuse existing server functions that read data and metadata from the diverse sources (cache, log, and disk storage). These func-

tions also encapsulate BlueFS-specific complexity such as the organization of data on disk (e.g., on-disk files are hashed and stored in a hierarchical directory structure organized by hash value to improve lookup performance). Due to this reuse, the code size of our transducers is relatively small. For example, a transducer that we wrote to search ID3 tags and return all MP3 files with a specific artist required only 239 lines of C code.

5.2 Application-specific resolution

The second type of sprocket performs application-specific resolution similar to that proposed by Kumar et al. for the Coda file system [14]. Like Coda, BlueFS uses optimistic concurrency and supports disconnected operation. Therefore, it is possible that concurrent updates may be made to a file by different clients. When this occurs, the user is normally asked to manually resolve the conflict. As anyone who has used CVS knows, manual conflict resolution is a tedious and error-prone process.

Kumar et al. observed that many types of files have an internal structure that can be used by the file system to *automatically* resolve conflicts. For example, if one client adds an artist to the ID3 tag of an MP3 file, while another client adds a rating for the song, a file system with knowledge of this data type can determine that the two updates are orthogonal. An automatic resolution for these two updates would produce a file that contains both the new artist and rating. However, like the transducer example in the previous section, BlueFS cannot perform such automatic resolution because it lacks the required knowledge about the data type.

To allow for automatic conflict resolution, we extended the conflict handling code in the BlueFS client daemon to allow for the optional invocation of a handler for specific data types. When the daemon tries to reintegrate an update that has been made on its client to the server, the server may detect that there has been a conflicting update made by another client (BlueFS stores a version number and the identifier of the last client to update the file in order to detect such conflicts). The client daemon then checks to see if there is a conflict handler registered for the data type (specifically, it checks to see if the name of the file matches a regular expression such as files that end in ".mp3"). If a match is found, the daemon invokes the sprocket registered for that data type.

Our original design had the sprocket do the entire resolution by reading and fetching the current version of the file stored at the server, comparing it to the version stored on the client, and then writing the result to a temporary file. However, this approach was unsatisfying for two reasons. First, it violated our rule that sprockets should never persistently change state. The design required the sprocket to communicate with the server, which is an externally visible event that changes persistent state on the server. The communication increments sequence numbers and perturbs the next message if the stream is encrypted. Second, the design did not promote reuse. Each resolution sprocket must separately implement code to fetch data from the server, read data from the client, and write the result to a temporary file.

Based on these observations, we refactored our design to perform resolution with two separate sprockets. The first sprocket determines the data to be fetched from the server; it returns this information as a list of data ranges. For example, an MP3 resolver would return the bytes that contain the ID3 tag. After executing the sprocket, the daemon fetches the required data. The first sprocket may be invoked iteratively to allow it to traverse data structures within a file. Thus, the work that is generic and that makes persistent changes to file system state is now done outside of the sprocket. A second benefit of this approach is that only a limited subset of a file's data needs to be fetched from the server; for large multimedia files, this substantially improves performance.

The daemon passes the second sprocket the range of data to examine that was output by the first sprocket, as well as the corresponding data in the client and server versions of the file to be resolved. The second sprocket performs the resolution and returns a *patch* that contains regions of data to add, delete, or replace in the server's version of the file. The daemon validates that the patch represents an internally consistent update to the file (e.g., that the bytes being deleted or replaced actually exist within the file). It sends the changes in the patch file to the server to complete the resolution. This design fits the sprocket model well since the format of the patch is well understood and can be validated by the file system before being applied; yet, the logic that generates the patch can be arbitrarily complex and reside within the sprocket. A bug in the sprocket could potentially produce an invalid ID3 header; however, since the application-specific metadata is opaque to the core file system, such a bug could not lead to a subsequent crash of the client daemon or server.

We have written an MP3 resolver that compares two ID3 tags and returns a new ID3 tag that merges concurrent updates from the two input tags. The first sprocket is invoked twice to determine the byte range of the ID3 tag in the file. The second sprocket performs the resolution and requests that the daemon replace the version of the ID3 tag at the server with a new copy that contains the merged updates. Typically, the patch contains a single entry that replaces the data in the original ID3 tag. However, if the size of the ID3 tag has grown, the patch may also request that additional bytes be inserted in the file after the location of the original ID3 tag. These two sprockets required a combined 474 lines of C code.

5.3 Device-specific processing

The final type of sprocket allows BlueFS to read and write the data stored on different types of consumer electronic devices. Prior to this work, BlueFS already allowed the user to treat devices such as iPods and digital cameras as clients of the distributed file system. Files on such devices are treated as replicas of files within the distributed namespace. When a consumer electronic device attaches to a general-purpose computer running the BlueFS client daemon, the daemon propagates changes made on the device to the distributed namespace of the file system. If the files in the distributed namespace have been modified since the device last attached to a BlueFS client, the daemon propagates those changes to the files on the device's local storage.

This previous support for consumer electronic devices assumed that all such devices export a generic file system interface through which BlueFS can read and write data. This is not true for all devices: for example, many cameras allow photos to be uploaded and downloaded using the Picture Transfer Protocol (PTP), and digital media players typically allow their data to be accessed through the UPnP Content Delivery Service (CDS). For each new type of interface, BlueFS must be extended to understand how to read, write, and search through the data on a device using its device-specific protocol.

The required functionality is akin to that of device drivers in modern operating systems. While the logic that allows consumer electronic devices to interact with the file system is generic, the particular interface used to read, write, and search through data on each device is often specific to the device type. We therefore chose to structure our code such that most functionality is implemented by a generic layer that calls into device-specific routines only when it needs to access data on the consumer electronic device. These low-level routines provide services such as listing the files on a device, reading data from each file, and creating new files on the device.

We have created two sets of device-specific routines: one for devices that export a file system interface, and one for cameras that use PTP. Other sets of routines could be added to expand the number of consumer electronic devices supported by BlueFS.

Potentially, we could have linked these interface routines directly into the file system daemon, in much the same way that device drivers are dynamically loaded into the kernel. However, we were cautioned by the poor reliability of device drivers in modern operating systems [25]. We felt that such software could be a substantial source of bugs, and we did not want faulty interface routines to have the capability to crash the file system or corrupt the data that it stores. Therefore, we implemented these interface routines as sprockets to isolate them from the rest of the file system.

For both the file system and PTP interface, we have created sprockets that implement functions that open files and directories, read them, and modify them. To improve performance, these sprockets are allowed to cache intermediary data in a temporary directory. They are also allowed to make system calls that interact with the specific device with which they are interfacing; for example, the PTP sprocket can communicate with the camera over the USB interface. These additional capabilities are allowed by expanding the whitelist for this particular sprocket type to enable the extended functionality. However, these sprockets are not allowed to make changes to data stored in BlueFS. Instead, they pass buffers to the file system daemon. The daemon validates the contents of each buffer before modifying the file system. We implemented the entire PTP sprocket interface using only 635 lines of C code.

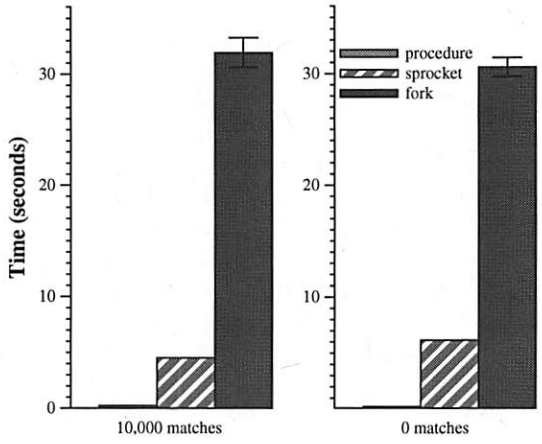
5.4 Other potential sprockets

Beyond the three types of sprockets we have already implemented, we see many more potential applications of sprockets in distributed file systems. One can insert sprockets into client and server code to collect statistics so that the file system can be tuned for better performance. Sprockets can be used to refine the results from directory listings — for example, if multimedia files are located on remote storage, they might be listed in a directory only if sufficient bandwidth is available to stream them from the remote source and play them without loss. Sprockets could also be used to support on-the-fly transcoding of data from one format to another. Sprockets could potentially implement application-specific caching policies: for instance, highly rated songs or movies that have been recorded but not yet viewed can be stored on mobile devices. In general, we believe that sprockets are a promising way to deal with the heterogeneity of the emerging class of consumer electronic devices, as well as the multimedia data formats that they support.

6 Evaluation

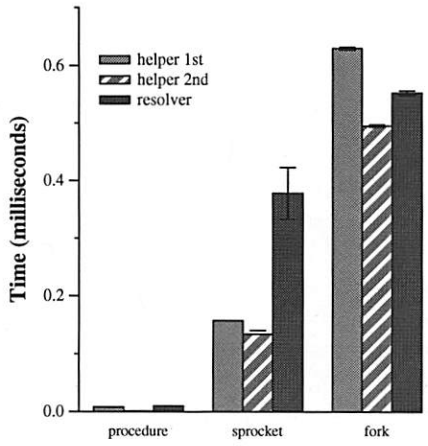
Our evaluation answers the following questions:

- What is the relative performance of extensions implemented through binary instrumentation, address-space sandboxing, and direct procedure calls?
- What are the effects of our binary instrumentation optimizations on performance?
- What isolation is provided for extensions implemented through binary instrumentation, address-space sandboxing, and direct procedure calls?



This figure compares the time to create a persistent query that lists MP3 songs by the band Radiohead using extensions implemented via procedure call, sprocket, and `fork`. The graph on the left shows the results when the file system contains 10,000 files, all of which match the persistent query; the graph on the right shows the results when none match. Each result is the mean of five trials — error bars show 90% confidence intervals.

Figure 2. Performance of the Radiohead transducer



This figure compares performance when resolving a conflict using an application-specific ID3 tag resolver using procedure call, sprocket, and fork-based implementations. A helper extension is invoked twice to determine which data needs to be resolved, and a resolver extension performs the actual resolution. Each bar shows the time to resolve conflicts with 100 files. Each result is the mean of five trials — error bars show 90% confidence intervals.

Figure 3. Application-speciÆc conØict resolution

6.1 Methodology

For our evaluation we used a single computer with a 3.02 GHz Pentium 4 processor and 1 GB of RAM — this computer acts as both a BlueFS client and server. The computer runs Red Hat Enterprise Linux 3 (kernel version 2.4.21-4). When a second BlueFS client is required, we add a IBM T20 laptop with a 700 MHz Pentium III processor and 128 MB of RAM connected over a 100 Mbps switch. The IBM T20 runs Red Hat Linux Enterprise 4 (kernel version 2.6.9-22). Each BlueFS client is configured with a 500 MB write log and does not cache data on disk. We used the PIN toolkit version 7259 compiled for gcc version 3.2. All results were measured using the `gettimeofday` system call.

6.2 Radiohead transducer

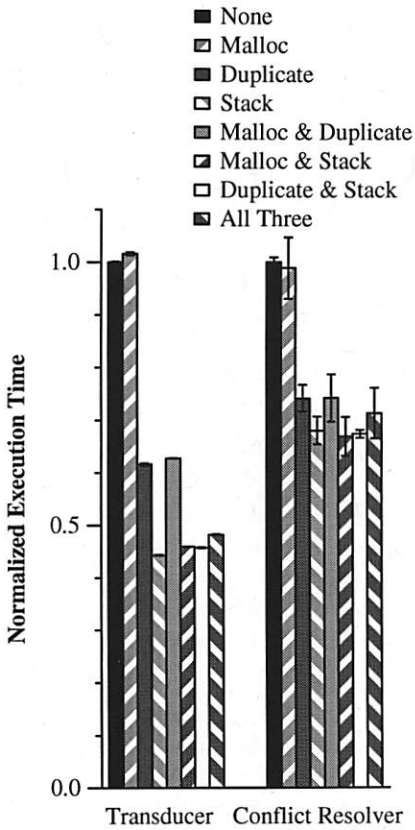
The first experiment measures the performance of a transducer extension that determines whether or not an MP3 has the artist tag "Radiohead", as described in Section 5.1. Figure 2 shows the performance of the extension in two different scenarios. In the left graph, the file system is first populated with 10,000 MP3 files with the ID3 tag designating Radiohead as the artist. The first bar in the graph shows the time to run the sprocket 10,000 times, once for each file in the file system, and generate the persistent query when the extension is executed as a function call inside the BlueFS address space. As expected, the function call implementation is extremely fast since it provides no isolation. The second bar shows performance running the extension as a sprocket, with PIN-based binary instrumentation providing isolation. The instrumentation slows the execution of the extension by a factor of 20 but ensures that a buggy sprocket will not adversely affect the server.

The last bar in the graph shows performance when executing the extension using `fork` as described in Section 3.3. While `fork` provides many of the same benefits as sprockets, its performance is over 6 times worse. For this small extension, the per-instruction performance cost of binary instrumentation is much cheaper than the constant performance cost of copying the file server's page table and flushing the TLB when executing `fork`.

The right graph in Figure 2 shows the performance of the Radiohead transducer when BlueFS is populated with 10,000 MP3 files, none of which are by the band Radiohead. Thus, the resulting persistent query will be empty. The results of the second experiment are similar to the first. However, the extension executes more code in this scenario because it checks for the possible existence of a version 2 ID3 tag when it finds that no version 1 ID3 tag exists. In the first experiment, the second check is never executed. The additional code has a proportionally greater affect on the sprocket implementation because of its high per-instruction cost.

6.3 Application-specific conflict resolution

The next experiment measures the performance of a set of extensions that resolve a conflict within the ID3 tag of an MP3. When a client sends an operation to the server (e.g., a file system write) that conflicts with the version at the server, the client invokes an extension to try to automatically resolve the conflict before requiring the user to manually intervene. We populated BlueFS with 100 MP3



This figure shows effects of combinations of our optimizations on the performance of our sprocket tests: the Radiohead transducer with 10,000 matches and a run of the application specific conflict resolver. Results are the average of five pre-instrumented executions with 90% confidence intervals and are normalized to the unoptimized performance.

Figure 4. Optimization performance

files, each of which is 3 MB in size. We then modified two different fields within the ID3 tag of each file on two different BlueFS clients. After ensuring that one client had reconciled its log with the server, we flushed the second client's log, creating a conflict in all 100 files. The client then invokes two extensions to resolve the conflict. The first, *helper*, extension is invoked twice to determine where the ID3 tag is located in the file. The first invocation reads the ID3 header, which determines the size of the rest of the tag; the second invocation reads the rest of the tag. The second, *resolver*, extension creates a patch that resolves the conflict. This process is repeated for each of the 100 files.

Figure 3 shows the performance of each implementation. The sprocket implementation is substantially faster than the fork-based implementation on all extensions, though the difference in performance is greater for the first two helper invocations (because they execute fewer instructions). The resolver extension is still faster with the sprocket implementation than with `fork`, but shows a substantially smaller advantage than the others. This is

optimization	transducer	conflict resolver
Malloc (total)	0.00%	2.78%
Duplicate (total)	82.44%	81.71%
Stack (total)	99.45%	93.35%
Malloc (unique)	0.00%	2.01%
Duplicate (unique)	0.38%	2.68%
Stack (unique)	17.39%	15.08%

This table shows the fraction of memory backups prevented by the three optimizations. The first three rows show the fraction of memory backups prevented by each optimization. The second three rows show the fraction of memory backups prevented by only that optimization and no other. Results are the average of five runs of a single execution of each extension.

Table 2. Effects of optimizations

because the resolver extension runs longer, causing the cumulative cost of executing instrumented code to approach the cost of `fork`. While the performance of binary instrumentation might improve with further code optimization, we believe that sprockets of substantially greater complexity than this one should probably be executed using `fork` for best performance.

6.4 Optimizations

Given this set of experiments, we next measured the effectiveness of our proposed binary instrumentation optimizations which eliminate saving and restoring data at addresses allocated by the sprocket, duplicated in the undo log, or in the section of the stack used by the sprocket. These three techniques are intended to improve performance by inserting an inexpensive check before each write performed by the sprocket that tests whether overwritten data needs to be saved in the undo log.

Since each optimization adds a test that is executed before each write, optimizations must provide a substantial reduction in logging to overcome the cost of testing. Figure 4 shows the time taken for both the Radiohead transducer and conflict resolver with combinations of optimizations turned on.

To understand these results, we first measured the fraction of writes each optimization prevents from creating an undo log entry. As shown in the upper half Table 2, avoiding either stack writes or duplicates of already logged addresses prevents almost all new log entries. For these sprockets, the malloc optimization is less effective; the Radiohead transducer does not use malloc and the conflict resolver performs few writes to the memory it allocates.

Seeing the large overlap in writes covered by these optimizations, we next investigated how much each contributed to the total reduction in logging. The lower half of Table 2 shows the fraction of writes that are uniquely covered by each optimization. In this view, the malloc

buggy sprocket	procedure result	fork result	sprocket result
memory leak	crash	correct	correct
memory stomp	crash	correct	correct
segfault	crash	extension terminated	extension terminated
file leak	crash	correct	correct
wrong close	hang	correct	extension terminated
infinite loop	hang	extension terminated	extension terminated
call exec	exec & exit	exec executed	extension terminated

This table shows the results when a buggy extension is executed under three different execution environments. "Correct" means that the sprocket completed successfully without a negative effect on the BlueFS file system. "Extension terminated" means a problem was detected and the extension halted without adversely affecting the file system or its data.

Table 1. Result of executing buggy extensions

optimization looks more useful as writes it covers are usually not covered by the other optimizations.

Since the Radiohead transducer sprocket does not use malloc, the malloc optimization simply imposes additional cost. For this sprocket, the stack optimization alone is the most effective; adding the duplicate optimization prevents an additional 0.38% of writes from creating undo log entries, but this benefit is less than the cost of its test on every write.

On the conflict resolver sprocket, the effects are somewhat different. Again, the stack optimization is the most effective. Adding the other optimizations produces no significant difference. This suggests that very simple tests, such as the malloc optimization, that test if the address to be logged is within a certain range, can break even if they prevent around 2% of writes from triggering logging.

6.5 When good sprockets go bad

Next, we implemented eight buggy extensions and observed how their execution affected the BlueFS server — the results are shown in Table 1. The first extension leaks memory by allocating a 10 MB buffer and returning without deallocating the buffer. When the extension is run as a function call, the file server crashes after repeated invocation when it runs out of memory. When `fork` is used, the child address space is reclaimed each time the sprocket exits, so there are no negative effects. Likewise, the sprocket implementation exhibits no negative effects due to its rollback of address space changes.

The second extension overwrites an important data structure in the BlueFS server address space (the pointer to the head of the write-ahead log) with `NULL`. As expected, the extension crashes the server when run as a function call and it has no effect when run using `fork`. When run as a sprocket, the extension does not affect the server because the memory stomp is undone after the extension completes.

Another common fault is an illegal access to memory that causes a `segfault`. The third extension creates this fault by dereferencing a pointer to an invalid memory location. If the extension is executed as a function call, the server is terminated. If the extension is run via `fork`, the child process dies as a result of the `segfault` and an error message is returned to the parent process. The sprocket infrastructure correctly catches the `segfault` signal and returns an error to the core file system.

Leaking file handle resources can also be problematic. We created an extension that opens a file but forgets to close it. When we ran this extension multiple times as a function call, the server eventually crashed due to the resource leak. With both the `fork` and sprocket implementations, the resource leak is prevented by the cleanup executed after the extension finishes executing.

A buggy extension might also close a descriptor that it did not open. We therefore created an extension that closed all open descriptors, even those it did not itself open, before it exits. Executing this extension as a procedure call disconnected all current clients of the file server and prevented them from reconnecting by closing the port on which the server listens for incoming connections. When the extension is run with `fork`, the server's file handles are not affected by the sprocket's mistake. When the the extension is run as a sprocket, the system call whitelist detects that the sprocket is trying to close a file descriptor that it did not open and aborts the sprocket. Alternatively, we could have chosen to ignore the `close` call altogether, but we felt that triggering an error return was the best way to handle this bug.

Another common danger is extension code that does not terminate. The sixth row of Table 1 shows results for an extension that executes an infinite loop. Running the sprocket multiple times via a function call causes the server to hang as it runs out of threads. With sprockets, a timer expiration triggers termination of the sprocket. A similar approach can be used to terminate the child process when using `fork`.

The last sprocket attempts to execute a new program by calling `exec`. When executed as a function call, the server simply ceases to exist since its address space is

replaced by that of another program. With sprockets, the system call whitelist detects that a sprocket is attempting a disallowed system call. The PIN tool immediately rolls back the sprocket's execution, and returns an error to the file system. The `fork` implementation allows the extension to `exec` the specified executable, which is probably not a desirable behavior.

7 Related work

To the best of our knowledge, sprockets are the first system to use binary instrumentation and a transactional model to allow arbitrary code to run safely inside a distributed file system's address space.

Our use of binary instrumentation to isolate faults builds on the work done by Wahbe et al. [26] to sandbox an extension inside a program's address space. However, instead of limiting access to the address space outside the sandbox, we provide unfettered access to the address space but record modifications and undo them when the extension completes execution.

VINO [22] used software fault isolation in the context of operating system extensions. However, VINO extensions do not have access to the full repertoire of kernel functionality and are prevented from accessing certain data. Permitted kernel functions are specified by a list. Those functions must check parameters sent to them by the extension to protect the kernel. In contrast, sprockets can call any function and access any data.

Nooks [25] used this technique for device drivers. The Exokernel [5] allowed user-level code to implement many services traditionally provided by the operating system. Rather than focus on kernel extensions, sprockets target functionality that is already implemented at user-level. This has several advantages, including the ability to access user-level tools and libraries. The sprocket model also introduces minimal changes to the system being extended because it requires little refactoring of core file system code and makes extensions appear as much like a procedure call as possible.

Type-safe languages are another approach to protection. The SPIN project [25] used the type-safe Modula-3 language to guarantee safe behavior of modules loaded into the operating system. However, this approach may require extra effort from the extension developer to express the desired functionality and limits the reuse of existing file system code, much of which is not currently implemented in type-safe languages. Languages can be taken even further [15] by allowing provable correctness in limited domains. However, this is not applicable to our style of extension which can perform arbitrary calculation.

Other methods for extending file system functionality such as Watchdogs [3] and stackable file systems [8, 12] provide safety, but operate through a more restrictive interface that allows extensions only at certain pre-defined VFS operations such as `open`, `close`, and `write`. The sprocket interface is not necessarily appropriate for such course-grained extensions; instead, we target fine-grained extensions that are a few hundred lines of source code at most.

The use of sprockets to evaluate file system state was inspired by the predicates used by Joshi et al. [11] to detect the exploitation of vulnerabilities through virtual machine introspection. Evaluating a predicate provides similar functionality to a transaction that is never committed. The evaluation of a sprocket has similar goals in that it extracts a result from the system without perturbing the system's address space. However, since the code we are isolating runs only at user level, we can provide the needed isolation by using existing operating system primitives instead of a virtual machine monitor.

Like projects on software [23] and hardware [9] transactional memory, sprockets rely on hiding changes to memory from other threads to ensure that all threads view consistent state. One transactional memory implementation, LogTM [18], also uses a log to record changes to memory state. In the future, it may be possible to improve the performance of sprockets, particularly on multicore systems, by leveraging these techniques.

8 Conclusion

Sprockets are designed to be a safe, fast, and easy-to-use method for extending the functionality of file systems implemented at user level. Our results are encouraging in many respects. We were able to implement every sprocket that we attempted in a few hundred lines of code. Our sprocket implementation using binary instrumentation caught several serious bugs that we introduced into extensions and allowed the file system to recover gracefully from programming errors. Sprocket performance for very simple extensions can be an order of magnitude faster than a `fork`-based implementation. Yet, we also found that there are upper limits to the amount of complexity that can be placed in a sprocket before binary instrumentation becomes more expensive than `fork`. Extensions that are more than several thousand lines of source code are probably better supported via address-space sandboxing.

In the future, we would like to explore this issue in greater detail, perhaps by creating an adaptive mechanism that could monitor sprocket performance and choose the best implementation for each execution. We would also like to explore the use of the whitelist to restrict sprocket functionality: since the whitelist is implemented using a PIN tool, we may be able to specify novel policies that restrict the particular data being passed to system calls rather than just what system calls

are allowed. In general, we believe that sprockets are a promising avenue for meeting the extensibility needs of current distributed file systems and may be suited to the needs of other domains such as integrated development environments and games.

Acknowledgments

We thank Manish Anand for suggestions that improved the quality of this paper. The work is supported by the National Science Foundation under awards CNS-0509093 and CNS-0306251. Jason Flinn is supported by NSF CAREER award CNS-0346686. Ed Nightingale is supported by a Microsoft Research Student Fellowship. Intel Corp and Motorola Corp have provided additional support. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, Motorola, the University of Michigan, or the U.S. government.

References

[1] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 95–109.

[2] BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E., FIUCZYNSKI, M., BECKER, D., CHAMBERS, C., AND EGGERS, S. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, Dec. 1995), pp. 267–284.

[3] BERSHAD, B. B., AND PINKERTON, C. B. Watchdogs - extending the unix file system. *Computer Systems 1*, 2 (Spring 1988).

[4] CHANG, F., AND GIBSON, G. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (New Orleans, LA, February 1999), pp. 1–14.

[5] ENGLER, D., KAASHOEK, M., AND J. O'TOOLE, J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), pp. 251–266.

[6] FUSE. Filesystem in userspace. `http://fuse.sourceforge.net/`.

[7] GIFFORD, D. K., JOUVELOT, P., SHELDON, M. A., AND O'TOOLE, J. W. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 16–25.

[8] HEIDEMANN, J. S., AND POPEK, G. J. File-system development with stackable layers. *ACM Transactions on Computer Systems 12*, 1 (1994), 58–89.

[9] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (May 1993), pp. 289–300.

[10] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[11] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 91–104.

[12] KHALIDI, Y. A., AND NELSON, M. N. Extensible file systems in spring. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, December 1993), pp. 1–14.

[13] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[14] KUMAR, P., AND SATYANARAYANAN, M. Flexible and safe resolution of file conflicts. In *Proceedings of the 1995 USENIX Winter Technical Conference* (New Orleans, LA, January 1995).

[15] LERNER, S., MILLSTEIN, T., RICE, E., AND CHAMBERS, C. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2005), ACM Press, pp. 364–377.

[16] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation* (Chicago, IL, June 2005), pp. 190–200.

[17] MAZIÈRES, D. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Technical Conference* (Boston, MA, June 2001), pp. 261–274.

[18] MOORE, K. E., BOBBA, J., MORAVAN, M. J., HILL, M. D., AND WOOD, D. A. Logtm: Log-based transactional memory. In *HPCA-12*.

[19] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 191–205.

[20] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the Blue File System. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 363–378.

[21] PEEK, D., AND FLINN, J. EnsemBlue: Integrating consumer electronics and distributed storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, November 2006), pp. 219–232.

[22] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (Seattle, Washington, October 1996), pp. 213–227.

[23] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Symposium on Principles of Distributed Computing* (1995), pp. 204–213.

[24] Spotlight overview. Tech. Rep. 2006-04-04, Apple Corp., Cupertino, CA, 2006.

[25] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, 2003), pp. 207–222.

[26] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, December 1993), pp. 203–216.

[27] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 273–288.

SafeStore: A Durable and Practical Storage System

Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin
The University of Texas at Austin

Abstract

This paper presents SafeStore, a distributed storage system designed to maintain long-term data durability despite conventional hardware and software faults, environmental disruptions, and administrative failures caused by human error or malice. The architecture of SafeStore is based on fault isolation, *which SafeStore applies aggressively along administrative, physical, and temporal dimensions by spreading data across autonomous storage service providers (SSPs). However, current storage interfaces provided by SSPs are not designed for high end-to-end durability. In this paper, we propose a new storage system architecture that (1) spreads data efficiently across autonomous SSPs using* informed hierarchical erasure coding *that, for a given replication cost, provides several additional 9's of durability over what can be achieved with existing black-box SSP interfaces, (2) performs an efficient end-to-end audit of SSPs to detect data loss that, for a 20% cost increase, improves data durability by two 9's by reducing MTTR, and (3) offers durable storage with cost, performance, and availability competitive with traditional storage systems. We instantiate and evaluate these ideas by building a SafeStore-based file system with an NFS-like interface.*

1 Introduction

The design of storage systems that provide data durability on the time scale of decades is an increasingly important challenge as more valuable information is stored digitally [10, 31, 57]. For example, data from the National Archives and Records Administration indicate that 93% of companies go bankrupt within a year if they lose their data center in some disaster [5], and a growing number of government laws [8, 22] mandate multi-year periods of data retention for many types of information [12, 50].

Against a backdrop in which over 34% of companies fail to test their tape backups [6] and over 40% of individuals do not back up their data at all [29], multi-decade scale durable storage raises two technical challenges. First, there exist a broad range of threats to data durability including media failures [51, 60, 67], software bugs [52, 68], malware [18, 63], user error [50, 59], administrator error [39, 48], organizational failures [24, 28], malicious insiders [27, 32], and natural disasters on the scale of buildings [7] or geographic regions [11]. Requiring robustness on the scale of decades magnifies them all: threats that could otherwise be considered negligible must now be addressed. Second, such a system has to be practical with cost, performance, and availability competitive with traditional systems.

Storage outsourcing is emerging as a popular approach to address some of these challenges [41]. By entrusting storage management to a Storage Service Provider (SSP), where "economies of scale" can minimize hardware and administrative costs, individual users and small to medium-sized businesses seek cost-effective professional system management and peace of mind vis-a-vis both conventional media failures and catastrophic events.

Unfortunately, relying on an SSP is no panacea for long-term data integrity. SSPs face the same list of hard problems outlined above and as a result even brand-name ones [9, 14] can still lose data. To make matters worse, clients often become aware of such losses only after it is too late. This opaqueness is a symptom of a fundamental problem: SSPs are separate administrative entities and the internal details of their operation may not be known by data owners. While most SSPs may be highly competent and follow best practices punctiliously, some may not. By entrusting their data to back-box SSPs, data owners may free themselves from the daily worries of storage management, but they also relinquish ultimate control over the fate of their data. In short, while SSPs are an economically attractive response to the costs and complexity of long-term data storage, they do not offer their clients any end-to-end guarantees on data durability, which we define as the probability that a specific data object will not be lost or

corrupted over a given time period.

Aggressive isolation for durability. SafeStore stores data redundantly across multiple SSPs and leverages diversity across SSPs to prevent permanent data loss caused by isolated administrator errors, software bugs, insider attacks, bankruptcy, or natural catastrophes. With respect to data stored at each SSP, SafeStore employs a "trust but verify" approach: it does not interfere with the policies used within each SSP to maintain data integrity, but it provides an *audit* interface so that data owner retain end-to-end control over data integrity. The audit mechanism can quickly detect data loss and trigger data recovery from redundant storage before additional faults result in unrecoverable loss. Finally, to guard data stored at SSPs against faults at the data owner site (e.g. operator errors, software bugs, and malware attacks), SafeStore restricts the interface to provide temporal isolation between clients and SSPs so that the latter export the abstraction of write-once-read-many storage.

Making aggressive isolation practical. SafeStore introduces an efficient storage interface to reduce network bandwidth and storage cost using an *informed hierarchical erasure coding* scheme, that, when applied across and within SSPs, can achieve near-optimal durability. SafeStore SSPs expose redundant encoding options to allow the system to efficiently divide storage redundancies across and within SSPs. Additionally, SafeStore limits the cost of implementing its "trust but verify" policy through an audit protocol that shifts most of the processing to the audited SSPs and encourages them proactively measure and report any data loss they experience. Dishonest SSPs are quickly caught with high probability and at little cost to the auditor using probabilistic spot checks. Finally, to reduce the bandwidth, performance, and availability costs of implementing geographic and administrative isolation, SafeStore implements a two-level storage architecture where a local server (possibly running on the client machine) is used as a soft-state cache, and if the local server crashes, SafeStore limits down-time by quickly recovering the critical meta data from the remote SSPs while the actual data is being recovered in the background.

Contributions. The contribution of this paper is a highly durable storage architecture that uses a new replication interface to distribute data efficiently across diverse set of SSPs and an effective audit protocol to check data integrity. We demonstrate that this approach can provide high durability in a way that is practical and economically viable with cost, availability, and performance competitive with traditional systems. We demonstrate these ideas by building and evaluating SSFS, an NFS-based SafeStore storage system. Overall, we show that SafeStore provides an economical alternative to realize multi-decade scale durable storage for individuals and small-to-medium sized businesses with limited resources. Note that although we focus our attention on outsourced SSPs, the SafeStore architecture could also be applied internally by large enterprises that maintain multiple isolated data centers.

2 Architecture and Design Principles

The main goal of SafeStore is to provide extremely durable storage over many years or decades.

2.1 Threat model

Over such long time periods, even relatively rare events can affect data durability, so we must consider broad range of threats along multiple dimensions—physical, administrative, and software.

Physical faults: Physical faults causing data loss include disk media faults [35, 67], theft [23], fire [7], and wider geographical catastrophes [11]. These faults can result in data loss at a single node or spanning multiple nodes at a site or in a region.

Administrative and client-side faults: Accidental misconfiguration by system administrators [39, 48], deliberate insider sabotage [27, 32], or business failures leading to bankruptcy [24] can lead to data corruption or loss. Clients can also delete data accidentally by, for example, executing "rm -r *". Administrator and client faults can be particularly devastating because they can affect replicas across otherwise isolated subsystems. For instance [27], a system administrator not only deleted data but also stole the only backup tape after he was fired, resulting in financial damages in excess of $10 million and layoff of 80 employees.

Software faults: Software bugs [52, 68] in file systems, viruses [18], worms [63], and Trojan horses can delete or corrupt data. A vivid example of threats due to malware is the recent phenomenon of ransomware [20] where an attacker encrypts a user's data and withholds the encryption key until a ransom is paid.

Of course, any of the listed faults may occur rarely. But at the scale of decades, it becomes risky to assume that no rare events will occur. It is important to note that some of these failures [7, 51, 60] are often correlated resulting in simultaneous data loss at multiple nodes while others [52] are more likely to occur independently.

Limitations of existing practice. Most existing approaches to data storage face two problems that are particularly acute in our target environments of individuals

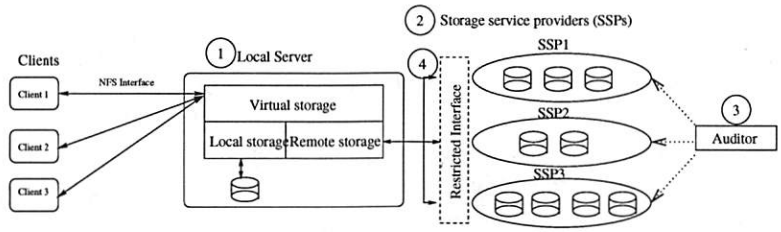


Fig. 1: SafeStore architecture

and small/medium businesses: (1) they depend too heavily on the operator or (2) they provide insufficient fault isolation in at least some dimensions.

For example, traditional removable-media-based-systems (e.g., tape, DVD-R) systems are labor intensive, which hurts durability in the target environments because users frequently fail to back their data up, fail to transport media off-site, or commit errors in the backup/restore process [25]. The relatively high risk of robot and media failures [3] and slow mean time to recover [44] are also limitations.

Similarly, although on-site disk-based [4, 16] backup systems speed backup/recovery, use reliable media compared to tapes, and even isolate client failures by maintaining multiple versions of data, they are vulnerable to physical site, administrative, and software failures.

Finally, network storage service providers (SSPs) [1, 2, 15, 21] are a promising alternative as they provide geographical and administrative isolation from users and they ride the technology trend of falling network and hardware costs to reduce the data-owner's effort. But they are still vulnerable to administrative failures at the service providers [9], organizational failures (e.g., bankruptcy [24, 41]), and operator errors [28]. They thus fail to fully meet the challenges of a durable storage system. We do, however, make use of SSPs as a component of SafeStore.

2.2 SafeStore architecture

As shown in Figure 1, SafeStore uses the following design principles to provide high durability by tolerating the broad range of threats outlined above while keeping the architecture practical, with cost, performance, and availability competitive with traditional systems.

Efficiency via 2-level architecture. SafeStore uses a two-level architecture in which the data owner's *local server* (①in Figure 1) acts as a cache and write buffer while durable storage is provided by multiple remote *storage service providers* SSPs ② The local server could be running on the client's machine or a different machine. This division of labor has two consequences. First, performance, availability, and network cost are improved because most accesses are served locally; we show this is crucial in Section 3. Second, management cost is improved because the requirements on the local system are limited (local storage is soft state, so local failures have limited consequences) and critical management challenges are shifted to the SSPs, which can have excellent economies of scale for managing large data storage systems [1, 26, 41].

Aggressive isolation for durability. We apply the principle of aggressive isolation in order to protect data from the broad range of threats described above.

- *Autonomous SSPs:* SafeStore stores data redundantly across multiple autonomous SSPs (②in Figure 1). Diverse SSPs are chosen to minimize the likelihood of common-mode failures across SSPs. For example, SSPs can be external commercial service providers [1, 2, 15, 21], that are geographically distributed, run by different companies, and based on different software stacks. Although we focus on *out-sourced* SSPs, large organizations can use our architecture with *in-sourced* storage across autonomous entities within their organization (e.g., different campuses in a university system.)
- *Audit:* Aggressive isolation alone is not enough to provide high durability as data fragment failures accumulate over time. On the contrary, aggressive isolation can adversely affect data durability because the data owner has little ability to enforce or monitor the SSPs' internal design or operation to ensure that SSPs follow best practices. We provide an end-to-end audit interface (③ in Figure 1) to detect data loss and thereby bound mean time to recover (MTTR), which in turn increases mean time to data loss (MTTDL). In Section 4 we describe our audit interface and show how audits limit the damage that poorly-run SSPs can inflict on overall durability.
- *Restricted interface:* SafeStore must minimize the likelihood that erroneous operation of one subsystem compromises the integrity of another [46]. In particular, because SSPs all interact with the local server, we must restrict that interface. For example, we must protect against careless users, malicious insiders, or

devious malware at the clients or local server that mistakenly delete or modify data. SafeStore's restricted SSP interface ④ provides temporal isolation via the abstraction of versioned write-once-read-many storage so that a future error cannot damage existing data.

Making isolation practical. Although durability is our primary goal, the architecture must still be economically viable.

- *Efficient data replication:* The SafeStore architecture defines a new interface that allows the local server to realize near-optimal durability using *informed hierarchical erasure coding* mechanism, where SSPs expose internal redundancy. Our interface does not restrict SSP's autonomy in choosing internal storage organization (replication mechanism, redundancy level, hardware platform, software stack, administrative policies, geographic location, etc.) Section 3 shows that our new interface and replication mechanism provides orders of magnitude better durability than *oblivious hierarchical encoding* based systems using existing black-box based interfaces [1, 2, 21].
- *Efficient audit mechanism:* To make audits of SSPs practical, we use a novel audit protocol that, like real world financial audits, uses self-reporting whereby auditor offloads most of the audit work to the auditee (SSP) in order to reduce the overall system resources required for audits. However, our audit takes the form of a challenge-response protocol with occasional spot-checks that ensure that an auditee that generates improper responses is quickly discovered and that such a discovery is associated with a cryptographic proof of misbehavior [30].
- *Other optimizations:* We use several optimizations to reduce overhead and downtime in order to make system practical and economically viable. First, we use a fast recovery mechanism to quickly recover from data loss at a local server where the local server comes online as soon as the meta-data is recovered from remote SSPs even while data recovery is going on in the background. Second, we use block level versioning to reduce storage and network overhead involved in maintaining multiple versions of files.

2.3 Economic viability

In this section, we consider the economic viability of our storage system architecture in two different settings, outsourced storage using commercial SSPs and federated storage using in-house but autonomous SSPs, and calibrate the costs by comparing with a less-durable local storage system.

We consider three components to storage cost: hardware resources, administration, and—for outsourced storage—profit. Table 1 summarizes our basic assumptions for a straw-man *Standalone* local storage system and for the local owner and SSP parts of a SafeStore system. In column B, we estimate the raw hardware and administrative costs that might be paid by an in-house SSP. We base our storage hardware costs on estimated full-system 5-year total cost of ownership (TCO) costs in 2006 for large-scale internet services such as Internet Archive [26]. Note that using the same storage cost for a large-scale, specialized SSP and for smaller data owners and Standalone systems is conservative in that it may overstate the relative additional cost of adding SSPs. For network resources, we base our costs on published rates in 2006 [17]. For administrative costs, we use Gray's estimate that highly efficient internet services require about 1 administrator to manage 100TB while smaller enterprises are typically closer to one administrator per 10TB but can range from one per 1TB to 1 per 100TB [49] (Gray notes, "But the real cost of storage is management" [49]). Note that we assume that by transforming local storage into a soft-state cache, SafeStore simplifies local storage administration. We therefore estimate local hardware and administrative costs at 1 admin per 100TB.

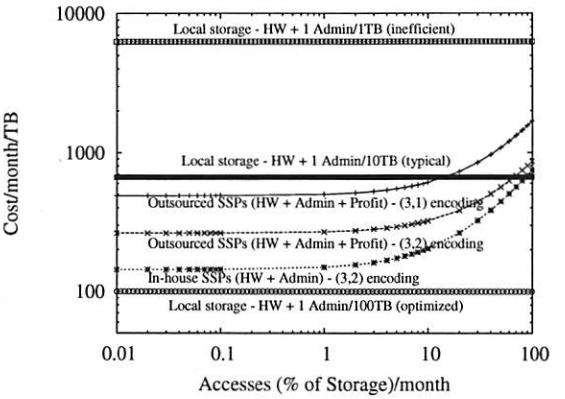


Fig. 2: Comparison of SafeStore cost v. accesses to remote storage (as a percentage of straw-man Standalone local storage) varies.

In Figure 2, the storage cost of in-house SSP includes SafeStore's hardware (cpu, storage, network) and administrative costs. We also plot the straw-man local storage system with 1, 10, or 100 TB per administrator. The outsourced SSP lines show SafeStore costs assuming SSPs prices include a profit by using Amazon's S3 storage service pricing. Three points stand out. First, additional replication to SSPs increases cost (as inter-SSP data encoding, as discussed in section 3, is raised from (3,2) to (3,1)), and the network cost rises rapidly as the remote access rate increases. These factors motivate SafeStore's architectural decisions to (1) use ef-

	Standalone	SafeStore In-house	SafeStore SSP (Cost+Profit)
Storage	$30/TB/month [26]	$30/TB/month [26]	$150/TB/month [1]
Network	NA	$200/TB [17]	$200/TB [1]
Admin	1 admin/[1,10,100]TB ([inefficient,typical,optimized]) [49]	1 admin/100TB [49]	Included [1]

Table 1: System cost assumptions. Note that a *Standalone* system makes no provision for isolated backup and is used for cost comparison only.

ficient encoding and (2) minimize network traffic with a large local cache that fully replicates all stored state. Second, when SSPs are able to exploit economies of scale to reduce administrative costs below those of their customers, SafeStore can reduce overall system costs even when compared to a less-durable Standalone local-storage-only system. Third, even for customers with highly-optimized administrative costs, as long as most requests are filtered by the local cache, SafeStore imposes relatively modest additional costs that may be acceptable if it succeeds in improving durability.

The rest of the paper is organized as follows. First, in Section 3 we present and and evaluate our novel *informed hierarchical erasure coding* mechanism. In Section 4, we address SafeStore's audit protocol. Later, in Section 5 we describe the SafeStore interfaces and implementation. We evaluate the prototype in Section 6. Finally, we present the related work in Section 7.

3 Data replication interface

This section describes a new replication interface to achieve near-optimal data durability while limiting the internal details exposed by SSPs, controlling replication cost, and maximizing fault isolation.

SafeStore uses hierarchical encoding comprising inter-SSP and intra-SSP redundancy: First, it stores data redundantly across different SSPs, and then each SSP internally replicates data entrusted to it as it sees fit. Hierarchical encoding is the natural way to replicate data in our setting as it tries to maximize fault-isolation across SSPs while allowing SSP's autonomy in choosing an appropriate internal data replication mechanism. Different replication mechanisms such as erasure coding [55], RAID [35], or full replication can be used to store data redundantly at inter-SSP and intra-SSP levels (any replication mechanism can be viewed as some form of (k,l) encoding [65] from durability perspective, where l out of k encoded fragments are required to reconstruct data). However, it requires proper balance between inter-SSP and intra-SSP redundancies to maximize end-end durability for a fixed storage overhead. For example, consider a system willing to pay an overall 6x redundancy cost using 3 SSPs with 8 nodes each. If, for example, each SSP only provides the option of (8,2) intra-SSP encoding, then we can use at most (3,2) inter-SSP encoding. This combination gives gives 4 9's less durability for the same overhead compared to a system that uses (3,1) encoding at the inter-SSP level and (8,4) encoding at the intra-SSP level at all the SSPs.

3.1 Model

The overall storage overhead to store a data object is $(n_0/m_0 + n_1/m_1 + ...n_{k-1}/m_{k-1})/l$, when a data object is hierarchically encoded using (k,l) erasure coding across k SSPs, and SSPs 0 through $k-1$ internally use erasure codings (n_0,m_0), (n_1,m_1),....(n_{k-1},m_{k-1}), respectively. We assume that the number of SSPs(k) is fixed and a data object is (possibly redundantly) stored at all SSPs. We do not allow varying k as it requires additional internal information about various SSPs (MTTF of nodes, number of nodes, etc.) which may not be available in order to choose optimal set of k nodes. Instead, we tackle the problem of finding optimal distribution of inter-SSP and intra-SSP redundancies for a fixed k. The end-to-end data durability can be estimated as a function of these variables using a simple analytical model, detailed in Appendix A of our extended report [45], that considers two classes of faults. *Node faults* (e.g. physical faults like sector failures, disk crashes, etc.) occur within an SSP and affect just one fragment of an encoded object stored at the SSP. *SSP faults* (e.g., administrator errors, organizational failures, geographical failures, etc.) are instead simultaneous or near-simultaneous failures that take out all fragments across which an object is stored within an SSP. To illustrate the approach, we consider a baseline system consisting of 3 SSPs with 8 nodes each. We use a baseline MTTDL of 10 years due to invidual node faults and 100 years for SSP failures and assume both are independent and identically distributed. We use MTTR of data of 2 days (e.g. to detect and replace a faulty disk) for node faults and 10 days for SSP failures. We use the probability of data loss of an object during a 10 year period to characterize durability because expressing end-to-end durability as MTTDL can be misleading [35] (although MTTDL can be easily computed from the probability of data loss as shown in our report [45]). Later, we change the distribution of nodes across SSPs, MTTDL and MTTR of node failures within SSPs, to model diverse SSPs. The conclusions that we draw here are general and not specific

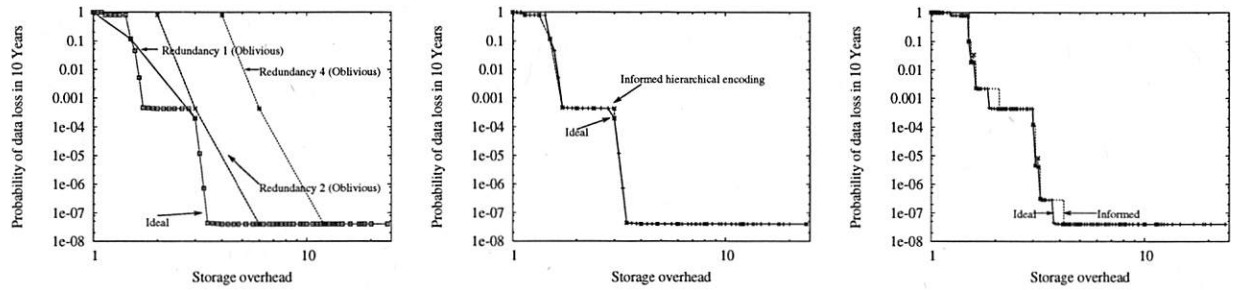


Fig. 3: (a) Durability with Black-box interface with fixed intra-SSP redundancy (b) Informed hierarchical encoding (c) Informed hierarchical encoding with non-uniform distribution

to this setup; we find similar trends when we change the total number of nodes, as well as MTTDL and MTTR of correlated *SSP faults*.

3.2 Informed hierarchical encoding

A client can maximize end-to-end durability if it can control both intra-SSP and inter-SSP redundancies. However, current black-box storage interfaces exported by commercial outsourced SSPs [1,2,21] do not allow clients to change intra-SSP redundancies. With such a black-box interface, clients perform *oblivious hierarchical encoding* as they control only inter-SSP redundancy. Figure 3(a) plots the optimal durability achieved by an *ideal* system that has full control of inter-SSP and intra-SSP redundancy and a system using *oblivious hierarchical encoding*. The latter system has 3 lines for different fixed intra-SSP redundancies of 1, 2, and 4, where each line has 3 points for each of the 3 different inter-SSP encodings((3,1), (3,2) and (3,3)) that a client can choose with such a black-box interface. Two conclusions emerge. First, for a given storage overhead, the probability of data loss of an *ideal* system is often orders of magnitude lower than a system using *oblivious hierarchical encoding*, which therefore is several 9's short of optimal durability. Second, a system using *oblivious hierarchical encoding* often requires 2x-4x more storage than *ideal* to achieve the same durability.

To improve on this situation, SafeStore describes an interface that allows clients to realize near-optimal durability using *informed hierarchical encoding* by exercising additional control on intra-SSP redundancies. With this interface, each SSP exposes the set of redundancy factors that it is willing to support. For example, an SSP with 4 internal nodes can expose redundancy factors of 1 (no redundancy), 1.33, 2, and 4 corresponding, respectively, to the (4,4), (4,3), (4,2) and (4,1) encodings used internally.

Our approach to achieve near-optimal end-to-end durability is motivated by the stair-like shape of the curve tracking the durability of *ideal* as a function of storage overhead (Figure 3(a)). For a fixed storage overhead, there is a tradeoff between inter-SSP and intra-SSP redundancies, as a given overhead O can be expressed as $1/l \times (r_0 + r_1 + ..r_{k-1})$, when (k,l) encoding is used across k SSPs in the system with intra-SSP redundancies of r_0 to r_{k-1} (where $r_i = n_i/m_i$). Figure 3(a) shows that durability increases dramatically (moving down one step in the figure) when inter-SSP redundancy increases, but does not improve appreciably when additional storage is used to increase intra-SSP redundancy beyond a threshold that is close to but greater than 1. This observation is backed by mathematical analysis in the extended report [45].

Hence, we propose a heuristic biased in favor of spending storage to maximize inter-SSP redundancy as follows:

- First, for a given number k of SSPs, we maximize the inter-SSP redundancy factor by minimizing l. In particular, for each SSP i, we choose the minimum redundancy factor $r'_i > 1$ exposed by i, and we compute l as $l = \lfloor (r'_0 + r'_1 + ...r'_{k-1})/O \rfloor$.
- Next, we distribute the remaining overhead $(O - 1/l \times (r'_0 + r'_1 + ..r'_{k-1}))$ among the SSPs to minimize the standard deviation of the intra-SSP redundancy factors r_i that are ultimately used by the different SSPs.

Figure 3(b) shows that this new approach, which we call *informed hierarchical coding*, achieves near optimal durability in a setting where three SSPs have the same number of nodes (8 each) and the same MTTDL and MTTR for internal node failures. These assumptions, however, may not hold in practice, as different SSPs are likely to have a different number of nodes, with different MTTDLs and MTTRs. Figure 3(c) shows the result of an experiment in which SSPs have a different number of nodes—and, therefore, expose different sets of redundancy factors. We still use 24 nodes, but we distribute them non-uniformly (14, 7, 3) across the

SSPs: informed hierarchical encoding continues to provide near-optimal durability. This continues to be true even when there is a skew in MTTDL and MTTR (due to node failures) across SSPs. For instance, Figure 4 uses the same non-uniform node distribution of Figure 3(c), but the (MTTDL, MTTR) values for node failures now differ across SSPs—they are, respectively, (10 years, 2 days), (5 years, 3 days), and (3 years, 5 days). Note that, by assigning the worst (MTTDL, MTTR) for node failures to the SSP with least number of nodes, we are considering a worst-case scenario for informed hierarchical encoding.

These results are not surprising in light of our discussion of Figure 3(a): durability depends mainly on maximizing inter-SSP redundancy and it is only slightly affected by the internal data management of individual SSPs. In our extended technical report [45] we perform additional experiments that study the sensitivity of informed hierarchical encoding to changes in the total number of nodes used to store data across all SSPs and in MTTDL and MTTR for SSP failures: they all confirm the conclusion that a simple interface that allows SSPs to expose the redundancy factors they support is all it is needed to achieve, through our simple informed hierarchical encoding mechanism, near optimal durability.

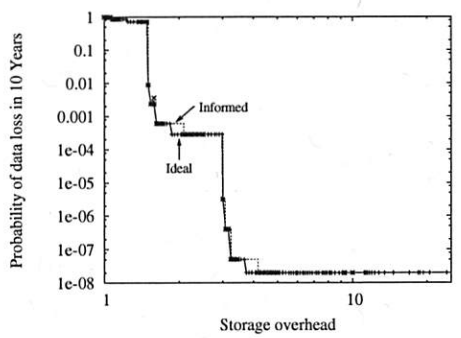


Fig. 4: Durability with different MTTDL and MTTR for node failures across SSPs

SSPs can provide such an interface as part of their SLA (service level agreement) and charge clients based on the redundancy factor they choose when they store a data object. The interface is designed to limit the amount of detail that an SSP must expose about the internal organization. For example, an SSP with 1000 servers each with 10 disks might only expose redundancy options (1.0, 1.1, 1.5, 2.0, 4.0, 10.0), revealing little about its architecture. Note that the proposed interface could allow a dishonest SSP to cheat the client by using less redundancy than advertised. The impact of such false advertising is limited by two factors: First, as observed above, our design is relatively insensitive to variations in intra-SSP redundancy. Second, the end to end audit protocol described in the next section limits the worst-case damage any SSP can inflict.

4 Audit

We need an effective audit mechanism to quickly detect data losses at SSPs so that data can be recovered before multiple component failures resulting in unrecoverable loss. An SSP *should* safeguard the data entrusted to it by following best practices like monitoring hardware health [62], spreading coded data across drives and controllers [35] or geographically distributed data centers, periodically scanning and correcting latent errors [61], and quickly notifying a data owner of any lost data so that the owner can restore the data from other SSPs and maintain a desired replication level. However, the principle of isolation argues against blindly assuming SSPs are flawless system designers and operators for two reasons. First, SSPs are separate administrative entities, and their internal details of operation may not be verifiable by data owners. Second, given the imperfections of software [18, 52, 68], operators [39, 48], and hardware [35, 67], even name-brand SSPs may encounter unexpected issues and silently lose customer data [9, 14]. Auditing SSP data storage embodies the end-to-end principle (in almost exactly the form it was first described) [58], and frequent auditing ensures a short Mean Time To Detect (MTTD) data loss, which helps limit worst-case Mean Time To Recover (MTTR). It is important to reduce MTTR in order to increase MTTDL as a good replication mechanism alone cannot improve MTTDL over a long time-duration spanning decades.

The technical challenge to auditing is to provide an end-to-end guarantee on data integrity while minimizing cost. These goals rule out simply reading stored data across the network as too expensive (see Figure 2) and, similarly, just retrieving a hash of the data as not providing an end-to-end guarantee (the SSP may be storing the hash not the data.). Furthermore, the audit protocol must work with data erasure-coded across SSPs, so a simple scheme that sends a challenge to multiple identical replicas and then compare the responses such as those in LOCKSS [46] and Samsara [37] do not work. We must therefore devise an inexpensive audit protocol despite the fact that no two replicas store the same data.

To reduce audit cost, SafeStore's audit protocol borrows a strategy from real-world audits: we push most of the work onto the auditee and ask the auditor to spot check the auditee's reports. Our reliance on self-reporting by SSPs drives two aspects of the protocol design. First, the protocol is believed to be *shortcut*

free–audit responses from SSPs are guaranteed to embody end-to-end checks on data storage– under the assumption that collision resistant modification detection codes [47] exist. Second, the protocol is *externally verifiable* and *non-repudiable*—falsified SSP audit replies are quickly detected (with high probability) and deliberate falsifications can be proven to any third party.

4.1 Audit protocol

The audit protocol proceeds in three phases: (1) data storage, (2) routine audit, and (3) spot check. Note that the auditor may be co-located with or separate from the owner. For example, audit may be outsourced to an external auditor when data owners are offline for extended periods. To authorize SSPs to respond to auditor requests, the owner signs a certificate granting audit rights to the auditor's public key, and all requests from the auditor are authenticated against such a certificate (these authentication handshakes are omitted in the description below.) We describe the high level protocol here and detail it in the report [45].

Data storage. When an object is stored at an SSP, the SSP signs and returns to the data owner a *receipt* that includes the object ID, cryptographic hash of the data, and storage expiration time. The data owner in turn verifies that the signed hash matches the data it sent and that the receipt is not malformed with an incorrect id or expiration time. If the data and hash fail to match, the owner retries sending the write message (data could have been corrupted in the transmission); repeated failures indicate a malfunctioning SSP and generate a notification to the data owner. As we detail in Section 5, SSPs do not provide a delete interface, so the expiration time indicates when the SSP will garbage collect the data. The data owner collects such valid receipts, encodes them, and spreads them across SSPs for durable storage.

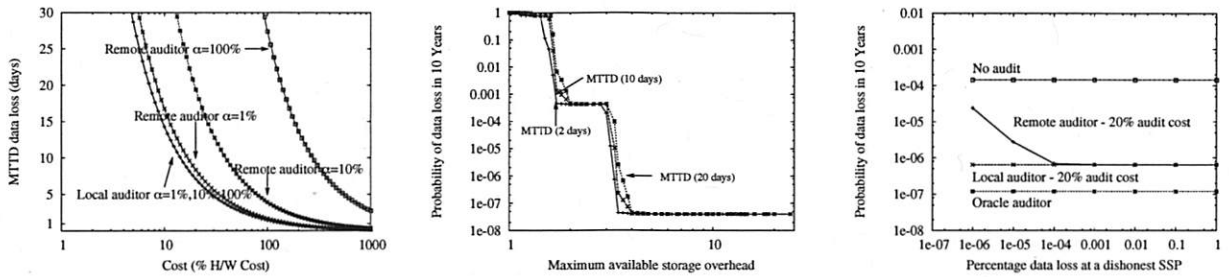
Routine audit. The auditor sends to an SSP a list of object IDs and a random challenge. The SSP computes a cryptographic hash on both the challenge and the data. The SSP sends a signed message to the auditor that includes the object IDs, the current time, the challenge, and the hash computed on the challenge and the data ($H(challenge + data_{objId})$). The auditor buffers the challenge responses if the messages are well-formed, where a message is considered to be well-formed if none of the following conditions are true: the signature does not match the message, the response with an unacceptably stale timestamp, the response with the wrong challenge, or the response indicates error code (e.g., he SSP detected data is corrupt via internal checks or the data has expired). If the auditor does not receive any response from the SSP or if it receives a malformed message, the auditor notifies the data owner, and the data owner reconstructs the data via cached state or other SSPs and stores the lost fragment again. Of course, the owner may choose to switch SSPs before restoring the data and/or may extract penalties under their service level agreement (SLA) with the SSP, but such decisions are outside the scope of the protocol.

We conjecture that the audit response is shortcut free: an SSP must possess object's data to compute the correct hash. An honest SSP verifies the data integrity against the challenge-free hash stored at the creation time before sending a well-formed challenge response. If the integrity check fails (data is lost or corrupted) it sends the error code for lost data to the auditor. However, a *dishonest* SSP can choose to send a syntactically well-formed audit response with bogus hash value when the data is corrupted or lost. Note that the auditor just buffers well-formed messages and does not verify the integrity of the data objects covered by the audit in this phase. Yet, routine audits serve two key purposes. First, when performed against honest SSPs, they provide end-to-end guarantees about the integrity of the data objects covered by the audit. Second, they force dishonest SSPs to produce a signed, non-repudiable statement about the integrity of the data objects covered by the audit.

Spot check. In each round, after it receives audit responses in the routine audit phase, the auditor randomly selects $\alpha\%$ of the objects to be spot checked. The auditor then retrieves each object's data (via the owner's cache, via the SSP, or via other SSPs) and verifies that the cryptographic hash of the challenge and data matches the challenge response sent by the SSP in the routine audit phase. If there is a mismatch, the auditor informs the data owner about the mismatch and provides the signed audit response sent by the SSP. The data owner then can create an externally-verifiable proof of misbehavior (POM) [45] against the SSP: the receipt, the audit response, and the object's data. Note that SafeStore local server encrypts all data before storing it to SSPs, so this proof may be presented to third parties without leaking the plaintext object contents. Also, note that our protocol works with erasure coding as the auditor can reconstruct the data to be spot checked using redundant data stored at other SSPs.

4.2 Durability and cost

In this section we examine how the low-cost audit protocol limits the damage from faulty SSPs. The SafeStore protocol specifies that SSPs notify data owners immediately of any data loss that the SSP cannot internally recover so that the owner can restore the desired replica-



(a) (b) (c)

Fig. 5: (a) Time to detect SSP data loss via audit with varying amounts of resources dedicated to audit overhead assuming honest SSPs. (b) Durability with varying MTTD. (c) Impact on overall durability with a dishonest SSP. The audit cost model for hardware, storage, and network bandwidth are described in [45]

tion level using redundant data. Figures 3 and 4 illustrate the durability of our system when the SSPs follow the requirement and immediately report failures. As explained below, Figure 5-(a) and (b) show that SafeStore still provides excellent data durability with low audit cost, if a data owner is unlucky and selects a *passive* SSP that violates the immediate-notify requirement and waits for an audit of an object to report that it is missing. Figure 5-(c) shows that if a data owner is really unlucky and selects a *dishonest* SSP that first loses some of the owner's data and then lies when audited to try to conceal that fact, the owner's data is still very likely to emerge unscathed. We evaluate our audit protocol with 1TB of data stored redundantly across three SSPs with inter-SSP encoding of (3,1) (the extended report [45] has results with other encodings).

First, assume that SSPs are *passive* and wait for an audit to check data integrity. Because the protocol uses relatively cheap processing at the SSP to reduce data transfers across the wide area network, it is able to scan through the system's data relatively frequently without raising system costs too much. Figure 5-(a) plots the mean time to detect data loss (MTTD) at a *passive* SSP as a function of the cost of hardware resources (storage, network, and cpu) dedicated to auditing, expressed as a percentage of the cost of the system's total hardware resources as detailed in the caption. We also vary the fraction of objects that are spot checked in each audit round (α) for both the cases with local (co-located with the data owner) and remote (separated over WAN) auditors. We reach following conclusions: (1) As we increase the audit budget we can audit more frequently and the time to detect data loss falls rapidly. (2) audit costs with local and remote auditors is almost the same when α is less than 1%. (3) The audit cost with local auditor does not vary much with increasing α (as there is no additional network overhead in retrieving data from the local data owner) whereas the audit cost for the remote auditor increases with increasing α (due to additional network overhead in retrieving data over the WAN). (4) Overall, if a system dedicates 20% of resources to auditing, we can detect a lost data block within a week (with a local or a remote auditor with $\alpha = 1\%$).

Given this information, Figure 5-(b) shows the modest impact on overall data durability of increasing the time to detect and correct such failures when we assume that all SSPs are *passive* and SafeStore relies on auditing rather than immediate self reporting to trigger data recovery.

Now consider the possibility of an SSP trying to brazen its way through an audit of data it has lost using a made-up value purporting to be the hash of the challenge and data. The audit protocol encourages rational SSPs that lose data to respond to audits honestly. In particular, we prove [45] that under reasonable assumptions about the penalty for an honest failure versus the penalty for generating a proof of misbehavior (POM), a rational SSP will maximize its utility [30] by faithfully executing the audit protocol as specified.

But suppose that through misconfiguration, malfunction, or malice, a node first loses data and then issues *dishonest* audit replies that claim that the node is storing a set of objects that it does not have. The spot check protocol ensures that if a node is missing even a small fraction of the objects, such cheating is quickly discovered with high probability. Furthermore, as that fraction increases, the time to detect falls rapidly. The intuition is simple: the probability of detecting a dishonest SSP in k audits is given by

$$p_k = 1 - (1-p)^k$$

where p is the probability of detection in an audit, which is given by

$$p = \frac{\sum_{i=1}^{m} \binom{n}{i}\binom{N-n}{m-i}}{\binom{N}{m}}, (\text{if } n \geq m)$$

WriteReceipt **write**(ID oid, byte data[], int64 size, int32 type, int64 expire);
ReadReply **read**(ID oid, int64 size, int32 type)
AttrReply **get_attr**(ID oid);
TTLReceipt **extend_expire**(ID oid, int64 expire);

Table 2: SSP storage interface

$$p = \frac{\sum_{i=1}^{n} \binom{m}{i}\binom{N-m}{n-i}}{\binom{N}{n}}, (\text{if } n < m)$$

where N is the total number of data blocks stored at an SSP, n is the number of blocks that are corrupted or lost and m is the number of blocks that are spot checked, α=(m/N) $\times$ 100.

Figure 5-(c) shows the overall impact on durability if a node that has lost a fraction of objects maximizes the time to detect these failures by generating *dishonest* audit replies. We fix the audit budget at 20% and measure the durability of SafeStore with local auditor (with α at 100%) as well as remote auditor (with α at 1%). We also plot the durability with *oracle detector* which detects the data loss immediately and triggers recovery. Note that the *oracle detector* line shows worse durability than the lines in Figure 5-(b) because (b) shows durability for a randomly selected 10-year period while (c) shows durability for a 10-year period that begins when one SSP has already lost data. Without auditing (*no audit*), there is significant risk of data loss reducing durability by three 9's compared to *oracle detector*. Using our audit protocol with *remote auditor*, the figure shows that a cheating SSP can introduce a non-negligible probability of small-scale data loss because it takes multiple audit rounds to detect the loss as it spot checks only 1% of data blocks. But that the probability of data loss falls quickly and comes closer to *oracle detector* line (with in one 9 of durability) as the amount of data at risk rises. Finally, with a *local auditor*, data loss is detected in one audit round independent of data loss percentage at the dishonest SSPs as a local auditor can spot check all the data. In the presence of dishonest SSPs, our audit protocol improves durability of our system by two 9's over a system with no audit at an additional audit cost of just 20%. We show in the extended report [45] that overall durability of our system improves with increasing audit budget and approaches the *oracle detector* line.

5 SSFS

We implement SSFS, a file system that embodies the SafeStore architecture and protocol. In this section, we first describe the SSP interface and our SSFS SSP implementation. Then, we describe SSFS's local server.

5.1 SSP

As Figure 1 shows, for long-term data retention SSFS local servers store data redundantly across administratively autonomous SSPs using erasure coding or full replication. SafeStore SSPs provide a simple yet carefully defined object store interface to local servers as shown in Table 2.

Two aspects of this interface are important. First, it provides non-repudiable receipts for writes and expiration extensions in order to support our spot-check-based audit protocol. Second, it provides *temporal isolation* to limit the data owner's ability to change data that is currently stored [46]. In particular, the SafeStore SSP protocol (1) gives each object an absolute expiration time and (2) allows a data owner to extend but not reduce an object's lifetime.

This interface supports what we expect to be a typical usage pattern in which an owner creates a ladder of backups at increasing granularity [59]. Suppose the owner wishes to maintain yearly backups for each year in the past 10 years, monthly backups for each month of the current year, weekly backups for the last four weeks, and daily backups for the last week. Using the local server's snapshot facility (see Section 5.2), on the last day of the year, the local server *writes* all current blocks that are not yet at the SSP with an expiration date 10-years into the future and also iterates across the most recent version of all remaining blocks and sends *extend_expire* requests with an expiration date 10-years into the future. Similarly, on the last day of each month, the local server writes all new blocks and extends the most recent version of all blocks; notice that blocks not modified during the current year may already have expiration times beyond the 1-year target, but these extensions will not reduce this time. Similarly, on the last day of each week, the local server writes new blocks and extends deadlines of the current version of blocks for a month. And every night, the local server writes new blocks and extends deadlines of the current version of all blocks for a week. Of course, SSPs ignore *extend_expire* requests that would shorten an object's expiration time.

SSP implementation. We have constructed a prototype SSFS SSP that supports all of the features described in this paper including the interface for servers and the interface for auditors. Internally, each SSP spreads data across a set nodes using erasure coding with a redundancy level specified for each data owner's account at account creation time.

For compatibility with legacy SSPs, we also implement a simplified SSP interface that allows data owners to store data to Amazon's S3 [1], which provides a sim-

ple non-versioned read/write/delete interface and which does not support our optimized audit protocol.

Issues. There are two outstanding issues in our current implementation. We believe all are manageable. .

First, in practice, it is likely that SSPs will provide some protocol for deleting data early. We assume that any such out-of-band early-delete mechanism is carefully designed to maximize resistance to erroneous deletion by the data owner. For concreteness, we assume that the payment stream for SSP services is well protected by the data owner and that our SSP will delete data 90 days after payment is stopped. So, a data owner can delete unwanted data by creating a new account, copying a subset of data from the old account to the new account, and then stopping payment on the old account. More sophisticated variations (e.g., using threshold-key cryptography to allow a quorum of independent administrators to sign off on a delete request) are possible.

Second, SSFS is vulnerable to resource consumption attacks: although an attacker who controls an owner's local server cannot reduce the integrity of data stored at SSPs, the attacker can send large amounts of long-lived garbage data and/or extend expirations farther than desired for large amounts of the owner's data stored at the SSP. We conjecture that SSPs would typically employ a quota system to bound resource consumption to within some budget along with an out-of-band early delete mechanism such as described in the previous paragraph to recover from any resulting denial of service attack.

5.2 Local Server

Clients interact with SSFS through a local server. The SSFS local server is a user level file system that exports the NFS 2.0 interface to its clients. The local server serves requests from local storage to improve the cost, performance, and availability of the system. Remote storage is used to store data durably to guard against local failures. The local server encrypts (using SHA1 and 1024 bit Rabin key signature) and encodes [55] (if data is not fully replicated) all data before sending it to remote SSPs, and it transparently fetches, decodes and decrypts data from remote storage if it is not present in the local cache.

All local server state except the encryption key and list of SSPs is soft state: given these items, the local server can recover the full filesystem. We assume both are stored out of band (e.g., the owner burns them to a CD at installation time and stores the CD in a safety deposit box).

Snapshots: In addition to the standard NFS calls, the SSFS local server provides a snapshot interface [16] that supports file versioning for achieving temporal isolation to tolerate client or administrator failures. A snapshot stores a copy in the local cache and also redundantly stores encrypted, erasure-coded data across multiple SSPs using the remote storage interface.

Local storage is structured carefully to reduce storage and performance overheads for maintaining multiple versions of files. SSFS uses block-level versioning [16, 53] to reduce storage overhead by storing only modified blocks in the older versions when a file is modified.

Other optimizations: SSFS uses a fast recovery optimization to recover quickly from remote storage when local data is lost due to local server failures (disk crashes, fire, etc.) The SSFS local server recovers quickly by coming online as soon as all metadata information (directories, inodes, and old-version information) is recovered and then fetching file data to fill the local cache in the background. If a missing block is requested before it is recovered, it is fetched immediately on demand from the SSPs. Additionally, local storage acts as a write-back cache where updates are propagated to remote SSPs asynchronously so that client performance is not affected by updates to remote storage.

6 Evaluation

To evaluate the practicality of the SafeStore architecture, we evaluate our SSFS prototype via microbenchmarks selected to stress test three aspects of the design. First, we examine performance overheads, then we look at storage space overheads, and finally we evaluate recovery performance.

In our base setup, client, local server, and remote SSP servers run on different machines that are connected by a 100 Mbit isolated network. For several experiments we modify the network to synthetically model WAN behavior. All of our machines use 933MHZ Intel Pentium III processors with 256 MB RAM and run Linux version 2.4.7. We use (3,2) erasure coding or full replication ((3,1) encoding) to redundantly store backup data across SSPs.

6.1 Performance

Figure 6 compares the performance of SSFS and a standard NFS server using the IOZONE [13] microbenchmark. In this experiment, we measure the overhead of SSFS's bookkeeping to maintain version information, but we do not take filesystem snapshots and hence no data is sent to the remote SSPs. Figure 6(a),(b), and (c) illustrates throughput for reads, throughput for synchronous and asynchronous writes, and throughput ver-

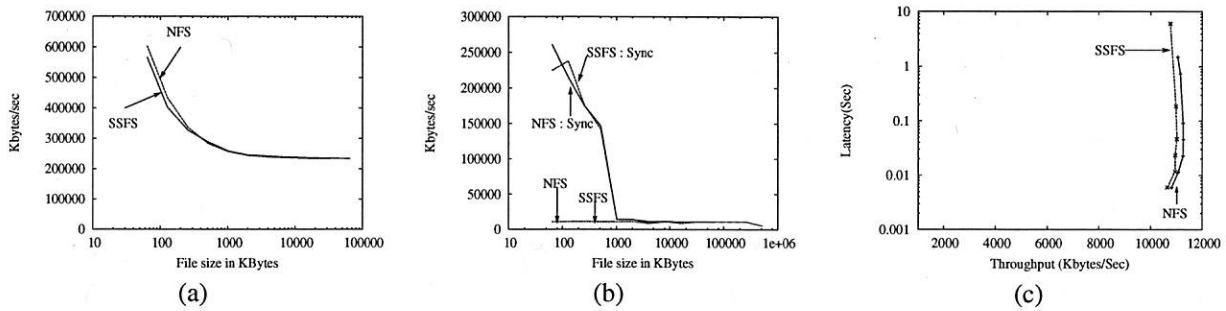


Fig. 6: IOZONE : (a) Read (b) Write (c) Latency versus Throughput

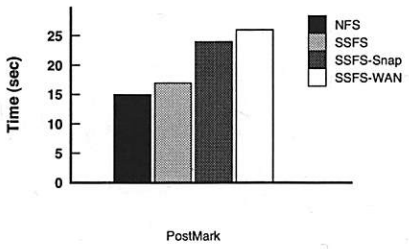


Fig. 7: Postmark: End-to-end performance

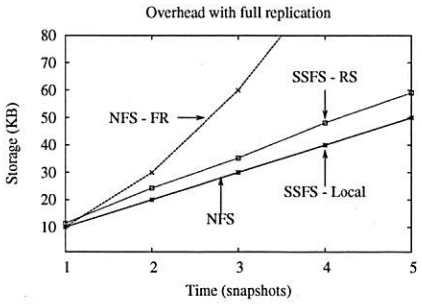


Fig. 8: Storage overhead

sus latency for SSFS and stand alone NFS. In all cases, SSFS's throughput is within 12% of NFS.

Figure 7 examines the cost of snapshots. Note SSFS sends snapshots to SSPs asynchronously, but we have not lowered the priority of these background transfers, so snapshot transfers can interfere with demand requests. To evaluate this effect, we add snapshots to the Postmark [19] benchmark, which models email/e-commerce workloads. The benchmark initially creates a pool of files and then performs a specified number of transactions consisting of creating, deleting, reading, or appending a file. We set file sizes to be between 100B and 100KB and run 50000 transactions. To maximize the stress on SSFS, we set the Postmark parameters to maximize the fraction of append and create operations. Then, we modify the benchmark to take frequent snapshots: we tell the server to create a new snapshot after every 500 transactions. As shown in the Figure 7, when no snapshots are taken SSFS takes 13% more time than NFS due to overhead involved in maintaining multiple versions. Turning on frequent snapshots increases the response time of SSFS (SSFS-snap in Figure 7) by 40% due to additional overhead due to signing and transmitting updates to SSPs. Finally, we vary network latencies to SSPs to study the impact of WAN latencies on performance when SSPs are geographically distributed over the Internet by introducing artificial delay (of 40 ms) at the SSP server. As shown in the Figure 7, SSFS-WAN response time increases by less than an additional 5%.

6.2 Storage overhead

Here, we evaluate the effectiveness of SSFS's mechanisms for limiting replication overhead. SSFS minimizes storage overheads by using a versioning system that stores the difference between versions of a file rather than complete copies [53]. We compare the storage overhead of SSFS's versioning file system and compare it with NFS storage that just keeps a copy of the latest version and also a naive versioning NFS file system (NFS-FR) that makes a complete copy of the file before generating a new version. Figure 8 plots the storage consumed by local storage (SSFS-LS) and storage at one remote server (SSFS-RS) when we use a (3,1) encoding. To expose the overheads of the versioning system, the microbenchmark is simple: we append 10KB to a file after every file system snapshot. SSFS's local storage takes a negligible amount of additional space compared to non-versioned NFS storage. Remote storage pays a somewhat higher overhead due to duplicate data storage when appends do not fall on block boundaries and due to additional metadata (integrity hashes, the signed write request, expiry time of the file, etc.)

The above experiments examine the case when the old and new versions of data have much in common and test whether SSFS can exploit such situations with low overhead. There is, of course, no free lunch: if there is little in common between a user's current data and

old data, the system must store both. Like SafeStore, Glacier uses a expire-then-garbage collect approach to avoid inadvertent file deletion, and their experience over several months of operation is that the space overheads are reasonable [40].

7 Related work

Several recent studies [31, 57] have identified the challenges involved in building durable storage system for multi-year timescales.

Flat erasure coding across nodes [33, 36, 40, 66] does not require detailed predictions of which sets of nodes are likely to suffer correlated failures because it tolerates any combinations of failures up to a maximum number of nodes. However, flat encoding does not exploit the opportunity to reduce replication costs when the system can be structured to make some failure combinations more likely than others. An alternative approach is to use *full replication* across sites that are not expected to fail together [43, 46], but this can be expensive.

SafeStore is architected to increase the likelihood that failures will be restricted to specific groups of nodes, and it efficiently deploys storage within and across SSPs to address such failures. Myriad [34] also argues for a 2-level (cross-site, within-site) coding strategy, but SafeStore's architecture departs from Myriad in keeping SSPs at arms-length from data owners by carefully restricting the SSP interface and by including provisions for efficient end-to-end auditing of black-box SSPs.

SafeStore is most similar in spirit to OceanStore [42] in that we erasure code indelible, versioned data across independent SSPs. But in pursuit of a more aggressive "nomadic data" vision, OceanStore augments this approach with a sophisticated overlay-based infrastructure for replication of location-independent objects that may be accessed concurrently from various locations in the network [54]. We gain considerable simplicity by using a local soft-state server through which all user requests pass and by focusing on storing data on a relatively small set of specific, relatively conventional SSPs. We also gain assurance in the workings of our SSPs through our audit protocol.

Versioning file systems [16, 50, 56, 59, 64] provide temporal isolation to tolerate client failures by keeping multiple versions of files. We make use of this technique but couple it with efficient, isolated, audited storage to address a broader threat model.

We argue that highly durable storage systems should audit data periodically to ensure data integrity and to limit worst-case MTTR. Zero-knowledge-based audit mechanisms [38, 47] are either network intensive or CPU intensive as their main purpose is to audit data without leaking any information about the data. SafeStore avoids the need for such expensive approaches by encrypting data before storing it. We are then able to offload audit duties to SSPs and probabilistically spot check their results. LOCKSS [46] and Samsara [37] audit data in P2P storage systems but assume that peers store full replicas so that they can easily verify if peers store identical data. SafeStore supports erasure coding to reduce costs, so our audit mechanism does not require SSPs to have fully replicated copies of data.

8 Conclusion

Achieving robust data storage on the scale of decades forces us to reexamine storage architectures: a broad range of threats that could be neglected over shorter timescales must now be considered. SafeStore aggressively applies the principle of *fault isolation* along administrative, physical, and temporal dimensions. Analysis indicates that SafeStore can provide highly robust storage and evaluation of an NFS prototype suggests that the approach is practical.

9 Acknowledgements

This work was supported in part by NSF grants CNS-0411026, CNS-0430510, and CNS-0509338 and by the Center for Information Assurance and Security at the University of Texas at Austin.

References

[1] Amazon S3 Storage Service. http://aws.amazon.com/s3.
[2] Apple Backup. http://www.apple.com.
[3] Concerns raised on tape backup methods. http://searchsecurity.techtarget.com.
[4] Copan Systems. http://www.copansys.com/.
[5] Data loss statistics. http://www.hp.com/sbso/serverstorage/protect.html.
[6] Data loss statistics. http://www.adrdatarecovery.com/content/adr_loss_stat.html.
[7] Fire destroys research center. http://news.bbc.co.uk/1/hi/england/hampshire/4390048.stm.
[8] Health Insurance Portability and Accountability Act (HIPAA). 104th Congress, United States of America Public Law 104-191.
[9] Hotmail incinerates customer files. http://news.com.com, June 3rd, 2004.
[10] "How much information ?". http://www.sims.berkeley.edu/projects/how-much-info/.
[11] Hurricane Katrina. http://en.wikipedia.org.
[12] Industry data retention regulations. http://www.veritas.com/van/articles/4435.jsp.
[13] IOZONE micro-benchmarks. http://www.iozone.org.
[14] Lost Gmail Emails and the Future of Web Apps. http://it.slashdot.org, Dec 29, 2006.
[15] NetMass Systems. http://www.netmass.com.
[16] Network Appliance. http://www.netapp.com.
[17] Network bandwidth cost. http://www.broadbandbuyer.com/formbusiness.htm.
[18] OS vulnerabilities. http://www.cert.com/stats.

[19] Postmark macro-benchmark. http://www.netapp.com/tech_library/postmark.html.

[20] Ransomware. http://www.networkworld.com/buzz/2005/092605-ransom.html.

[21] Remote Data Backups. http://www.remotedatabackup.com.

[22] Sarbanes-Oxley Act of 2002. 107th Congress, United States of America Public Law 107-204.

[23] Spike in Laptop Thefts Stirs Jitters Over Data. Washington Post, June 22, 2006.

[24] SSPs: RIP. Byte and Switch, 2002.

[25] Tape Replacement Realities. http://www.enterprisestrategygroup.com/ESGPublications.

[26] The Wayback Machine. http://www.archive.org/web/hardware.php.

[27] US secret service report on insider attacks. http://www.sei.cmu.edu/about/press/insider-2005.html.

[28] Victims of lost files out of luck. http://news.com.com, April 22, 2002.

[29] "data backup no big deal to many, until...". http://money.cnn.com, June 2006.

[30] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. of SOSP '05*, pages 45–58, Oct. 2005.

[31] M. Baker, M. Shah, D.S.Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. In *EuroSys*, 2006.

[32] L. Bassham and W. Polk. Threat assessment of malicious code and human threats. Technical report, National Institute of Standards and Technology Computer Security Division, 1994. http://csrc.nist.gov/nistir/threats/.

[33] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker. Total Recall: System support for automated availability management. In *Proceedings of 1st NSDI*, CA, 2004.

[34] F. Chang, M. Ji, S. T. A. Leung, J. MacCormick, S. E. Perl, and L. Zhang. Myriad: Cost-effective disaster tolerance. In *Proceeedings of FAST*, 2002.

[35] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID : High-performance,reliable secondary storage. *ACM Comp. Surveys*, 26(2):145–185, June 1994.

[36] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital Libraries*, San Fransisco, CA, Aug 1999.

[37] L. Cox and B. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proc. of SOSP03*.

[38] P. Golle, S. Jarecki, and I. Mironov. Cryptographic primitives enforcing communication and storage complexity. In *Financial Cryptography (FC 2002)*, volume 2357 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2003.

[39] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Trans. on Reliability*, 39(4):409–418, Oct. 1990.

[40] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Higly durable, decentralized storage despite massive correlated failures. In *Proceedings of 2nd NSDI*, CA, March 2004.

[41] R. Hassan, W. Yurcik, and S. Myagmar. The evolution of storage service providers. In *StorageSS'05*, VA,USA, 2005.

[42] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ASPLOS*, 2000.

[43] F. Junquiera, R. Bhagwan, K. Marzullo, S. Savage, and G. M. Voelker. Surviving internet catastrophes. In *Proceedings of the Usenix Annual Technical Conference*, April 2005.

[44] K. Keeton and E. Anderson. A backup appliance composed of high-capacity disk drives. In *HP Laboratories SSP Technical Memo HPL-SSP-2001-3*, April 2001.

[45] R. Kotla, L. Alvisi, and M. Dahlin. Safestore: A durable and practical storage system. Technical report, University of Texas at Austin, 2007. UT-CS-TR-07-20.

[46] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, M. Baker, and Y. Muliadi. Lockss: A peer-to-peer digital preservation system. *ACM Transactions on Computer Systems*, 23(1):2–50, Feb. 2005.

[47] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Hndbook of Applied Cryptography*. CRC Press, 2001.

[48] D. Openheimer, A. Ganapathi, and D. Patterson. Why do internet systems fail, and what can be done about it. In *Proceedings of 4th USITS*, Seattle,WA, March 2003.

[49] D. Patterson. A conversation with jim gray. *ACM Queue*, pages vol. 1, no. 4, June 2003.

[50] Z. Peterson and R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Trans. on Storage*, 1(2):190–212, May. 2005.

[51] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *Proceeedings of FAST*, 2007.

[52] V. Prabhakaran, L. Bairavasundaram, N. Agrawal, H. G. A. Arpaci-Dusseau, and R. Arpaci-Dusseau. IRON file systems. In *Proc. of SOSP '05*, 2005.

[53] K. M. Reddy, C. P. Wright, A. Hammer, and E. Zadok. A Versatile and user-oriented versioning file system. In *FAST*, 2004.

[54] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *FAST03*, Mar. 2003.

[55] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Comp. Comm. Review*, 27(2), 1997.

[56] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[57] D. S. H. Rosenthal, T. S. Robertson, T. Lipkis, V. Reich, and S. Morabito. Requirements for digital preservation systems: A bottom-up approach. *D-Lib Magazine*, 11(11), Nov. 2005.

[58] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM TOCS*, Nov. 1984.

[59] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of 17th ACM Symp. on Operating Systems Principles*, December 1999.

[60] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proceeedings of FAST*, 2007.

[61] T. Schwarz, Q. Xin, E. Miller, D. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *Proc. MASCOTS*, Oct. 2004.

[62] Seagate. Get S.M.A.R.T for reliability. Technical Report TP-67D, Seagate, 1999.

[63] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of 6th OSDI*, 2004.

[64] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in a comprehensive versioning file system. In *Proc. of FAST 2003*.

[65] H. Weatherspoon and J. Kubiatowicz. Erasure Coding versus replication: A quantitative comparison. In *Proceedings of IPTPS*, Cambridge,MA, March 2002.

[66] J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliccote, and P. K. Khosla. Survivable information storage systems. *IEEE Computer*, 33(8):61–68, Aug. 2000.

[67] Q. Xin, T. Schwarz, and E. Miller. Disk infant mortality in large storage systems. In *Proc of MASCOTS '05*, 2005.

[68] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of 6th OSDI*, December 2004.

POTSHARDS: Secure Long-Term Storage Without Encryption

Mark W. Storer Kevin M. Greenan Ethan L. Miller
University of California, Santa Cruz
Kaladhar Voruganti
Network Appliance†

Abstract

Users are storing ever-increasing amounts of information digitally, driven by many factors including government regulations and the public's desire to digitally record their personal histories. Unfortunately, many of the security mechanisms that modern systems rely upon, such as encryption, are poorly suited for storing data for indefinitely long periods of time—it is very difficult to manage keys and update cryptosystems to provide secrecy through encryption over periods of decades. Worse, an adversary who can compromise an archive need only wait for cryptanalysis techniques to catch up to the encryption algorithm used at the time of the compromise in order to obtain "secure" data.

To address these concerns, we have developed POTSHARDS, an archival storage system that provides long-term security for data with very long lifetimes without using encryption. Secrecy is achieved by using provably secure secret splitting and spreading the resulting shares across separately-managed archives. Providing availability and data recovery in such a system can be difficult; thus, we use a new technique, approximate pointers, in conjunction with secure distributed RAID techniques to provide availability and reliability across independent archives. To validate our design, we developed a prototype POTSHARDS implementation, which has demonstrated "normal" storage and retrieval of user data using indexes, the recovery of user data using only the pieces a user has stored across the archives and the reconstruction of an entire failed archive.

1 Introduction

Many factors motivate the need for secure long-term archives, ranging from the relatively short-term (for archival purposes) requirements on preservation, retrieval and security properties demanded by recent legislation [1, 20] to the indefinite lifetimes of cultural and family heritage data. As users increasingly create and store images, video, family documents, medical records and legal records digitally, the need to securely preserve this data for future generations grows correspondingly. This information often needs to be stored securely; data such as medical records and legal documents that could be important to future generations must be kept indefinitely but must not be publicly accessible.

The goal of a secure, long-term archive is to provide security for relatively static data with an indefinite lifetime. There are three primary security properties that such archives aim to provide. First, the data stored must only be viewable by authorized readers. Second, the data must be available and accessible to authorized users within a reasonable amount of time, even to those who might lack a specific key. Third, there must be a way to confirm the integrity of the data so that a reader can be reasonably assured that the data that is read is the same as the data that was written.

The usage model of secure, long-term archival storage is write-once, read-maybe, and thus stresses throughput over low-latency performance. This is quite different from the top storage tier of a hierarchical storage solution that stresses low-latency access or even bottom-tier backup storage. The usage model of long-term archives also has the unique property that the reader may have little knowledge of the system's contents and no contact with the original writer; while file lifetimes may be indefinite, user lifetimes certainly are not. For digital "time capsules" that must last for decades or even centuries, the writer is assumed to be gone soon after the data has been written.

There are many novel storage problems [3, 32] that result from the potentially indefinite data lifetimes found in long-term storage. This is partially due to mechanisms such as cryptography that work well in the short-term but are less effective in the long-term. In long-term applications, encryption introduces the problems of lost

†Work performed while a member of IBM Almaden Research

keys, compromised keys and even compromised cryptosystems. Additionally, the management of keys becomes difficult because data will experience many key rotations and cryptosystem migrations over the course of several decades; this must all be done without user intervention because the user who stored the data may be unavailable. Thus, security for archival storage must be designed explicitly for the unique demands of long-term storage.

To address the many security requirements for long-term archival storage, we designed and implemented POTSHARDS (Protection Over Time, Securely Harboring And Reliably Distributing Stuff), which uses three primary techniques to provide security for long-term storage. The first technique is secret splitting [28], which is used to provide secrecy for the system's contents. Secret splitting breaks a block into n pieces, m of which must be obtained to reconstitute the block; it can be proven that any set of fewer than m pieces contains *no* information about the original block. As a result, secret splitting does not require the same updating as encryption, which is only computationally secure. By providing data secrecy without the use of encryption, POTSHARDS is able to move security from encryption to the more flexible and secure authentication realm; unlike encryption, authentication need not be done by computer, and authentication schemes can be easily changed in response to new vulnerabilities. Our second technique, *approximate pointers*, makes it possible to reconstitute the data in a reasonable time even if all indices over a user's data have been lost. This is achieved without sacrificing the secrecy property provided by the secret splitting. The third technique is the use of secure, distributed RAID techniques across multiple independent archives. In the event that an archive fails, the data it stored can be recovered without the need for other archives to reveal their own data.

We implemented a prototype of POTSHARDS and conducted several experiments to test its performance and resistance to failure. The current, CPU-bound implementation of POTSHARDS can read and write data at 2.5–5 MB/s on commodity hardware but is highly parallelizable. It also survives the failure of an entire archive with no data loss and little effect seen by users. In addition, we demonstrated the ability to rebuild a user's data from all of the user's stored shares without the use of a user index. These experiments demonstrate the system's suitability to the unique usage model of long-term archival storage.

2 Background

Since POTSHARDS was designed specifically for secure, long-term storage, we identified three basic design tenets to help focus our efforts. First, we assumed that encrypted data could be read by anyone given sufficient CPU cycles and advances in cryptanalysis. This means that, if all of an archives encrypted contents are obtained, an attacker can recover the original information. Second, data must be recoverable without any information from outside the set of archives. Thus, fulfilling requests in a reasonable time cannot require anything stored outside the archives, including external indexes or encryption keys. If this assumption is violated, there is a high risk that data will be unrecoverable after sufficient time has passed because the needed external information has been lost. Third, we assume that individuals are more likely to be malicious than an aggregate. In other words, our system can trust a group of archives even though it may not trust an individual archive. The chances of every archive in the system colluding maliciously is small; thus, we designed the system to allow rebuilding of stored data if all archives cooperate.

In designing POTSHARDS to meet these goals, we used concepts from various research projects and developed additional techniques. There are many existing storage systems that satisfy some of the design tenets discussed above, ranging from general-purpose distributed storage systems to distributed content delivery systems, to archival systems designed for short-term storage and archival systems designed for very specific uses such as public content delivery. A representative sample of these systems is summarized in Table 1. The remainder of this section discusses each of these primary tenets within the context of the related systems. Since these existing systems were not designed with secure, archival storage in mind, none has the combination of long-term data security and proof against obsolescence that POTSHARDS provides.

2.1 Archival Storage Models

Storage systems such as Venti [23] and Elephant [26] are concerned with archival storage, but tend to focus on the near-term time scale. Both systems are based on the philosophy that inexpensive storage makes it feasible to store many versions of data. Other systems, such as Glacier [13], are designed to take advantage of the underutilized client storage of a local network. These systems, and others that employ "checkpoint-style" backups, address neither the security concerns of the data content nor the needs of long-term archival storage. Venti and commercial systems such as the EMC Centera [12] use content-based storage techniques to achieve their goals, naming blocks based on a secure hash of their data. This approach increases reliability by providing an easy way to verify the content of a block against its name. As with the short-term storage systems described above, security

System	Secrecy	Authorization	Integrity	Blocks for Compromise	Migration
FreeNet	encryption	none	hashing	1	access based
OceanStore	encryption	signatures	versioning	m (out of n)	access based
FarSite	encryption	certificates	Merkle trees	1	continuous relocation
Publius	encryption	password (delete)	retrieval based	m (out of n)	
SNAD / Plutus	encryption	encryption	hashing	1	
GridSharing	secret splitting		replication	1	
PASIS	secret splitting		repair agents, auditing	m (out of n)	
CleverSafe	information dispersal	unknown	hashing	m (out of n)	none
Glacier	user encryption	node auth.	signatures	n/a	
Venti	none		retrieval	n/a	
LOCKSS	none		vote based checking	n/a	site crawling
POTSHARDS	secret splitting	pluggable	algebraic signatures	$O(R^{m-1})$	device refresh

Table 1: *Capability overview of the storage systems described in Section 2. "Blocks to compromise" lists the number of data blocks needed to brute-force recover data given advanced cryptanalysis; for POTSHARDS, we assume that an approximate pointer points to R shard identifiers. "Migration" is the mechanism for automatic replication or movement of data between nodes in the system.*

is ensured by encrypting data using standard encryption algorithms.

Some systems, such as LOCKSS [18] and Intermemory [10], are aimed at long-term storage of open content, preserving digital data for libraries and archives where file consistency and accessibility are paramount. These systems are developed around the core idea of very long-term access for public information; thus file secrecy is explicitly not part of the design. Rather, the systems exchange information about their own copies of each document to obtain consensus between archives, ensuring that a rogue archive does not "alter history" by changing the content of a document that it holds.

2.2 Storage Security

Many storage systems seek to enforce a policy of secrecy for their contents. Two common mechanisms for enforcing data secrecy are encryption and secret splitting.

2.2.1 Secrecy via Encryption

Many systems such as OceanStore [25], FARSITE [2], SNAD [19], Plutus [16], and e-Vault [15] address file secrecy but rely on the explicit use of keyed encryption. While this may work reasonably well for short-term secrecy needs, it is less than ideal for the very long-term security problem that POTSHARDS is addressing. Encryption is only computationally secure and the struggle between cryptography and cryptanalysis can be viewed as an arms race. For example, a DES encrypted message was considered secure in 1976; just 23 years later, in 1999, the same DES message could be cracked in under a day [29]; future advances in quantum computing have the potential to make many modern cryptographic algorithms obsolete.

The use of long-lived encryption implies that re-encryption must occur to keep pace with advances in cryptanalysis in order to ensure secrecy. To prevent a single archive from obtaining the unencrypted data, re-encryption must occur over the old encryption, resulting in a long key history for each file. Since these keys are all external data, a problem with any of the keys in the key history can render the data inaccessible when it is requested.

Keyed cryptography is only computationally secure, so compromise of an archive of encrypted data is a potential problem regardless of the encryption algorithm that is used. An adversary who compromises an encrypted archive need only wait for cryptanalysis techniques to catch up to the encryption used at the time of the compromise. If an insider at a given archive gains access to all of its data, he can decrypt any desired information even if the data is subsequently re-encrypted by the archive, since the insider will have access to the new key by virtue of his internal access. This is unacceptable, since the data's existence on a secure, long-term archive suggests that data will still be valuable even if the malicious user must wait several years to read it.

Some content publishing systems utilize encryption, but its use is not motivated solely by secrecy. Publius [34] utilizes encryption for write-level access control. Freenet [6] is designed for anonymous publication and encryption is used for plausible deniability over the contents of a users local store. As with secrecy, the use of encryption to enforce long-lived policy is problematic due to the mechanism's computationally secure nature.

2.2.2 Secrecy via Splitting

To address the issues resulting from the use of encryption, several recent systems including PASIS [11, 36] and GridSharing [33] have used or suggested the use [31] of *secret splitting* schemes [5, 22, 24, 28]; a related approach used by Mnemosyne [14] and CleverSafe [7] uses encryption followed by information dispersal (IDA) to attempt to gain the same security. In secret splitting, a secret is distributed by splitting it into a set number n of

shares such that no group of k shares ($k < m \leq n$) reveals any information about the secret; this approach is called an (m,n) threshold scheme. In such a scheme, any m of the n shares can be combined to reconstitute the secret; combining fewer than m shares reveals *no* information. A simple example of an (n,n) secret splitting scheme for a block B is to randomly generate $X_0,\ldots,X_{n-2}$, where $|X_i| = |B|$, and choose X_{n-1} so that $X_0 \oplus \cdots \oplus X_{n-2} \oplus X_{n-1} = B$. Secret splitting satisfies the second of our three tenets—data can be rebuilt without external information—but it can have the undesirable side-effect of combining the secrecy and redundancy aspects of the systems. Although related, these two elements of security are, in many respects, orthogonal to one another. Combining these elements also risks introducing compromises into the system by restricting the choices of secret splitting schemes.

To ensure that our third design tenet is satisfied, a secure long-term storage system must ensure that an attempt to breach security will be noticed by *somebody*, ensuring that the trust placed in the collection of archives can be upheld. Existing systems do not meet this goal because the secret splitting and data layout schemes they use are minimally effective against an inside attacker that knows the location of each of the secret shares. None of PASIS, CleverSafe, or GridSharing are designed to prevent attacks by insiders at one or more sites who can determine which pieces they need from other sites and steal those specific blocks of data, enabling a breach of secrecy with relatively minor effort. This problem is particularly difficult given the long time that data must remain secret, since such breaches could occur over years, making detection of small-scale intrusions nearly impossible. PASIS addressed the issue of refactoring secret shares [35]; however, this approach could compromise data in the system because the refactoring process may reveal information during the reconstruction process that a malicious archive could use to recover user data. By keeping this on separate nodes, the PASIS designers hoped to avoid information leakage. Mnemosyne used a local steganographic file system to hide chunks of data, but this approach is still vulnerable to rapid information leakage if the encryption algorithm is compromised because the IDA provides no additional protection to the distributed pieces.

2.3 Disaster Recovery

With long data lifetimes, hardware failure is a given; thus, dealing with a failed archive is inevitable. In addition, a long-term archival storage solution that relies upon multiple archives must be able to survive the loss of an archive for other reasons, such as business failure. Recovering from such large-scale disasters has long been a concern for storage systems [17]. To address this issue, systems such as distributed RAID [30], Myriad [4] and OceanStore [25] use RAID-style algorithms or more general redundancy techniques including (m,n) error correcting codes along with geographic distribution to guard against individual site failure. Secure, long-term storage adds the requirement that the secrecy of the distributed data must be ensured at all times, including during disaster recovery scenarios.

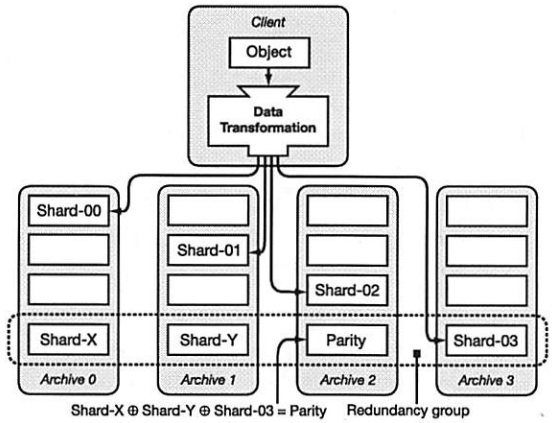


Figure 1: *An overview of POTSHARDS showing the data transformation component producing shards from objects and distributing them to independent archives. The archives utilize distributed RAID algorithms to securely recover shards in the event of a failure.*

3 System Overview

POTSHARDS is structured as a client communicating with a number of independent archives. Though the archives are independent, they assist each other through distributed RAID techniques to protect the system from archive loss. POTSHARDS stores user data by first splitting it into secure *shards*. These shards are then distributed to a number of archives, where each archive exists within its own security domain. The read procedure is similar but reversed; a client requests shards from archives and reconstitutes the data.

Data is prepared for storage during ingestion by a *data transformation* component that transforms *objects* into a set of secure shards which are distributed to the archives, as shown in Figure 1; similarly, this component is also responsible for reconstituting objects from shards during extraction. The data transformation component runs on a system **separate** from the archives on which the shards reside, and can fulfill requests from either a single client or many clients, depending on the implementation. This approach provides two benefits: the data never reaches an archive in an unsecured form; and multiple CPU-bound data transformation processes can generate shards in parallel for a single set of physical archives.

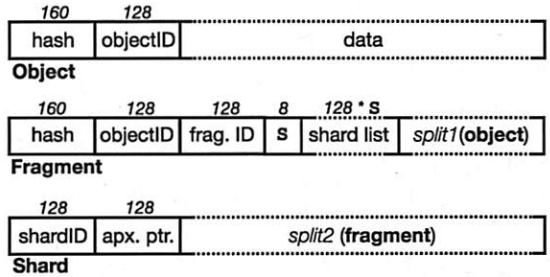


Figure 2: *Data entities in POTSHARDS, with size (in bits) indicated above each field. Note that entities are not shown to scale relative to one another. S is the number of shards that the fragment produces. split1 is an XOR secret split and split2 is a Shamir secret split in POTSHARDS.*

The archives operate in a manner similar to financial banks in that they are relatively stable and they have a incentive (financial or otherwise) to monitor and maintain the security of their contents. Additionally, the barrier to entry for a new archive should be relatively high (although POTSHARDS does takes precautions against a malicious insider); security is strengthened by distributing shards amongst the archives, so it is important that each archive can demonstrate an ability to protect its data. Other benefits of archive independence include reducing the effectiveness of insider attacks and making it easier to exploit the benefits of geographic diversity in physical archive locations. For these reasons, a single entity, such as a multinational company, should still maintain multiple independent archives to gain these security and reliability benefits.

3.1 Data Entities and Naming

There are three main data objects in POTSHARDS: *objects*, *fragments* and *shards*. As Figure 1 shows, objects contain the data that users submit to the system at the top level. Fragments are used within the data transformation component during the production of shards, which are the pieces actually stored on the archives. The details of these data entities can be seen in Figure 2.

All data entities in the current implementation of POTSHARDS are given unique 128-bit identifiers. The first 40 bits of the name uniquely identify the client in the same manner as a bank account is identified by an account number. The remaining 88 bits are used to identify the data entity. The length of the identifier could be extended relatively easily in future implementations. The names for entities that do not directly contribute to security within POTSHARDS, such as those for objects, can be generated in any way desired. However, the security and recovery time for a set of shards is directly related to the shards' names; thus, shards' IDs must be chosen with great care to ensure a proper density of names, providing sufficient security.

In addition to uniquely identifying data entities within the system, IDs play an important role in the secret splitting algorithms used in POTSHARDS. For secret splitting techniques that rely on linear interpolation [28], the reconstitution algorithm must know the original order of the secret shares. Knowing the order of the shards in a shard tuple can greatly reduce the time taken to reconstitute the data by avoiding the need to try each permutation of share ordering. Currently, this ordering is done by ensuring that the numerical ordering of the shard IDs reflects the input order to the reconstitution algorithm.

3.2 Secrecy and Reliability Techniques

POTSHARDS utilizes three primary techniques in the creation and long-term storage of shards. First, secret splitting algorithms provide file secrecy without the need to periodically update the algorithm. This is due to the fact that perfect secret splitting is information-theoretically secure as opposed to only computationally secure. Second, approximate pointers between shards allow objects to be recovered from only the shards themselves. Thus, even if all indices over a user's shards are lost, their data can be recovered in a reasonable amount of time. Third, secure, distributed RAID techniques across multiple independent archives allow data to be recovered in the event of an archive failure without exposing the original data during archive reconstruction.

POTSHARDS provides data secrecy through the use of secret splitting algorithms; thus, there is no need to maintain key history because POTSHARDS does not use traditional encryption keys.Additionally, POTSHARDS utilizes secret splitting in a way that does not combine the secrecy and redundancy parameters. Storage of the secret shares is also handled in a manner that dramatically reduces the effectiveness of insider attacks. By using secret splitting techniques, the secrecy in POTSHARDS has a degree of future-proofing built into it—it can be proven that an adversary with infinite computational power cannot gain any of the original data, even if an entire archive is compromised. While not strictly necessary, the introduction of a small amount of redundancy at the secret splitting layer allows POTSHARDS to handle transient archive unavailability by not requiring that a reader obtain *all* of the shards for an object; however, redundancy at this level is used primarily for short-term failures.

POTSHARDS provides approximate pointers to enable the reasonably quick reconstitution of user data without any information that exists outside of the shards themselves. POTSHARDS users normally keep indexes allowing them to quickly locate the shards that they need to reconstitute a particular object, as described in Section 4.3, so normal shard retrieval consists of asking

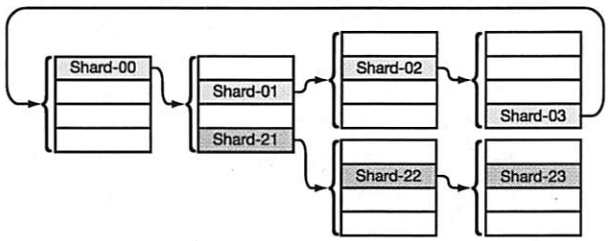
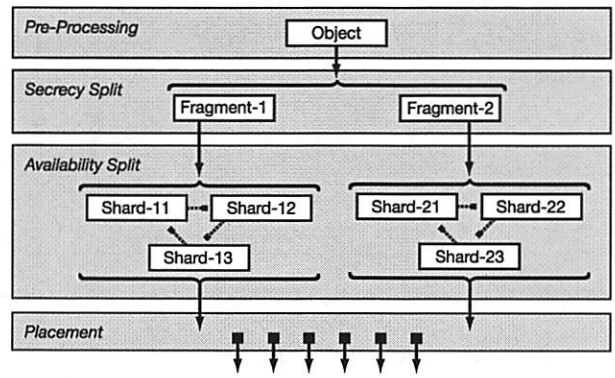


Figure 3: *Approximate pointers point to R "candidate" shards ($R = 4$ in this example) that might be next in a valid shard tuple. $Shards_{0X}$ make up a valid shard tuple. If an intruder mistakenly picks $shard_{21}$, he will not discover his error until he has retrieved sufficient shards and validation fails on the reassembled data.*

archives for the specific shards that make up an object, and is relatively fast. Approximate pointers are used when these user indexes are lost or otherwise unavailable. Since POTSHARDS can be used as a time capsule to secure data, it is foreseeable that a future user may be able to access the shards that they have a legal right to but have no idea how to combine them. The shards that can be combined together to reconstitute data form a *shard tuple*; an approximate pointer indicates the region in the user's private namespace where the next shard in the shard tuple exists, as shown in Figure 3. An approximate pointer has the benefit of making emergency data regeneration tractable while still making it difficult for an adversary to launch a targeted attack. If exact pointers were used, an adversary would know exactly which shards to target to rebuild an object. On the other hand, keeping no pointer at all makes it intractable to combine the correct shards without outside knowledge of which shards to combine. With approximate pointers, an attacker with one shard would only know the *region* where the next shard exists. Thus, a brute force attack requesting *every* shard in the region would be quite noticeable because the POTSHARDS namespace is intentionally kept sparse and an attack would result in requests for shards that do not exist. Unlike an index relating shards to objects that users would keep (and not store in the clear on an archive), an approximate pointer is part of the shard and is stored on the archive.

The archive layer in which the shards are stored consists of independent archives utilizing secure, distributed RAID techniques to provide reliability. As Figure 1 shows, archive-level redundancy is computed across sets of *unrelated* shards, so redundancy groups provide no insight into shard reassembly. POTSHARDS includes two novel modifications beyond the distributed redundancy explored earlier [4, 30]. The first is a secure reconstruction procedure, described in Section 4.2.1, that allows a failed archive's data to be regenerated in a manner that prevents archives from obtaining additional shards during the reconstruction; shards from the failed archive



(a) Four data transformation layers in POTSHARDS.

Module	*Input*	*Output*
Pre-processing	file	object
Secrecy split	object	set of fragments
Availability split	fragment	set of shards
Placement	set of shards	msgs for archives

(b) Inputs and outputs for each transformation layer.

Figure 4: *The transformation component consists of four levels. Approximate pointers are utilized at the second secret split. Note that locating one shard tuple provides no information about locating the shards from other tuples.*

are rebuilt only at the new archive that is replacing it. Second, POTSHARDS uses algebraic signatures [27] to ensure intra-archive integrity as well as inter-archive integrity. Algebraic signatures have the desirable property that the parity of a signature is the same as the signature of the parity, which can be used to prove the existence of data on other archives without revealing the data.

4 Implementation Details

This section details the components of POTSHARDS and how each contributes to providing long-term, secure storage. We first describe the transformation that POTSHARDS performs to ensure data secrecy. Next, we detail the inter-archive techniques POTSHARDS uses to provide long-term reliability. We then describe index construction; the use of indices makes "normal" data retrieval much simpler. Finally, we describe how we use approximate pointers to recover data with no additional information beyond the shards themselves, thus ensuring that POTSHARDS archives will be readable by future generations.

4.1 Data Transformation: Secrecy

Before being stored at the archive layer, user data travels through the data transformation component of POTSHARDS. This component is made up of four layers as shown in Figure 4.

1. **The pre-processing layer** divides files into fixed-sized,

b-byte objects. Additionally, objects include a hash that is used to confirm correct reconstitution.

2. **A secret splitting layer tuned for secrecy** takes an object and produces a set of fragments.
3. **A secret splitting layer tuned for availability** takes a fragment and produces a tuple of shards. It is also at this layer that the approximate pointers between the shards are created.
4. **The placement layer** determines how to distribute the shards to the archives.

4.1.1 Secret Splitting Layers

Fragments are generated at the first level of secret splitting, which is tuned for secrecy. Currently we use an XOR-based algorithm that produces n fragments from an object. To ensure security, the random data required for XOR splitting can be obtained through a physical process such as radio-active decay or thermal noise. As Figure 2 illustrates, fragments also contain metadata including a hash of the fragment's data which can be used to confirm a successful reconstitution.

A tuple of shards is produced from a fragment using another layer of secret splitting. This second split is tuned for availability which allows reconstitution in the event that an archive is down or unavailable when a request is made. In this version of POTSHARDS, shards are generated from a fragment using an (m,n) secret splitting algorithm [24, 28]. As the Figure 2 shows, shards contain no information about the fragments that they make up.

The two levels of secret splitting provide three important security advantages. First, as Figure 4 illustrates, the two-levels of splitting can be viewed as a tree with an increased fan out compared to one level of splitting. Thus, even if an attacker is able to locate all of the members of a shard tuple they can only rebuild a fragment and they have no information to help them find shards for the other fragments. Second, it separates the secrecy and availability aspects of the system. With two levels of secret splitting we do not need to compromise one aspect for the other. Third, it allows useful metadata to be stored with the fragments as this data will be kept secret by the second level of splitting. The details of shards and fragments are shown in Figure 2.

One cost of two-level secret splitting is that the overall storage requirements for the system are increased. A two-way XOR split followed by a $(2,3)$ secret split increases storage requirements by a factor of six; distributed RAID further increases the overhead. If a user desires to offset this cost, data can be submitted in a compressed archival form [37]; compressed data is handled just like any other type of data.

4.1.2 Placement Layer

The placement layer determines which archive will store each shard. The decision takes into account which shards belong in the same tuple and ensures that no single archive is given enough shards to recover data.

This layer contributes to security in POTSHARDS in four ways. First, since it is part of the data transformation component, no knowledge of which shards belong to an object need exist outside of the component. Second, the effectiveness of a insider attack at the archives is reduced because no single archive contains enough shards to reconstitute any data. Third, the effectiveness of an external attack is decreased because shards are distributed to multiple archives, each of which can exist in their own security domain. Fourth, the placement layer can take into account the geographic location of archives in order to maximize the availability of data.

4.2 Archive Design: Reliability

Storage in POTSHARDS is handled by a set of independent archives that store shards, actively monitor their own security and actively question the security of the other archives. The archives do not know which shards form a tuple, nor do they have any information about fragments or object reconstitution. A compromised archive does not provide an adversary with enough shards to rebuild user data. Nor does it provide an adversary with enough information to know where to find the appropriate shards needed to rebuild user data. Absent such precautions, the archive model would likely weaken the strong security properties provided by the other system components.

Since POTSHARDS is designed for long-term storage, it is inevitable that disasters will occur and archive membership will change over time. To deal with the threat of data loss from these events, POTSHARDS utilizes distributed RAID techniques. This is accomplished by dividing each archive into fixed-sized blocks and requiring all archives to agree on distributed, RAID-based methods over these blocks. Each block on the archive holds either shards or redundancy data.

When shards arrive at an archive for storage, ingestion occurs in three steps. First, a random block is chosen as the storage location of the shard. Next, the shard is placed in the last available slot in the the block. Finally, the corresponding parity updates are sent to the proper archives. Each parity update contains the data stored in the block and the appropriate parity block location. The failure of any parity update will result in a roll-back of the parity updates and re-placement of the shard into another block. Although it is assumed that all of the archives are trusted, we are currently analyz-

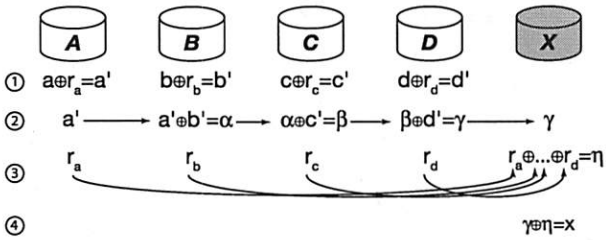


Figure 5: *A single round of archive recovery in a RAID 5 redundancy group. Each round consists of multiple steps. Archive N contains data n and generates random blocks r_n.*

ing the security effects of passing shard data between the archives during parity updates and exploring techniques for preventing archives from maliciously accumulating shards.

The distributed RAID techniques used in POTSHARDS are based on those from existing systems [4, 30]. In such systems, cost-effective, fault-tolerant, distributed storage is achieved by computing parity across unrelated data in wide area redundancy groups. Given an (n,k) erasure code, a redundancy group is an ordered set of k data blocks and $n-k$ parity blocks where each block resides on one of n distinct archives. The redundancy group can survive the loss of up to $n-k$ archives with no data loss. The current implementation of POTSHARDS has the ability to use Reed-Solomon codes or single parity to provide flexible and space-efficient redundancy across the archives.

POTSHARDS enhances the security of existing distributed RAID techniques through two important additions. First, the risk of information leakage during archive recovery is greatly mitigated through secure reconstruction techniques. Second, POTSHARDS utilizes algebraic signatures [27] to implement a secure protocol for both storage verification and data integrity checking.

4.2.1 Secure Archive Reconstruction

Reconstruction of data can pose a significant security risk because it can involve many archives and considerable amounts of data passing between archives. The secure recovery algorithm implemented within POTSHARDS exploits the independence of the archives participating in a redundancy group and the commutativity of evaluating the parity. Our reconstruction algorithm permits each archive to independently reconstruct a block of failed data without revealing any information about its data. The commutativity of the reconstruction procedure results in a reconstruction protocol that can occur in permutations, which greatly decreases the likelihood of successful collusion during archive recovery.

The recovery protocol begins with the confirmation of a partial or whole archive failure and, since each archive is a member of one or more redundancy groups, proceeds one redundancy group at a time. If a failure is confirmed, the archives in the system must agree on the destination of recovered data. A fail-over archive is chosen based on two criteria: the fail-over archive must not be a member of the redundancy group being recovered and it must have the capacity to store the recovered data. Due to these constraints multiple fail-over archives may be needed to perform reconstruction and redistribution. Future work will include ensuring that the choice of fail-over archives prevent any archive from acquiring enough shards to reconstruct user data.

Once the fail-over archive is selected, recovery occurs in multiple rounds. A single round of our secure recovery protocol over a single redundancy group is illustrated in Figure 5. In this example, the available members of a redundancy group collaborate to reconstruct the data from a failed archive onto a chosen archive X. An archive (which cannot be the fail-over and cannot be one of the collaborating archives) is appointed to manage the protocol by rebuilding one block at a time through multiple rounds of chained requests. A request contains an ordered list of archives, corresponding block identifiers and a data buffer and proceeds as follows at each archive in the chain:

1. Request α involving local block n arrives at archive N.
2. The archive creates a random block r_n and computes $n \oplus r_n = n'$.
3. The archive computes $\beta = \alpha \oplus n'$ and removes its entry from the request
4. The archive sends r_n directly to archive X.
5. β is sent to the next archive in the list.

This continues at each archive until the chain ends at archive X and the block is reconstructed. The commutativity of the rebuild process allows us to decrease the likelihood of data exposure by permuting the order of the chain in each round. This procedure is easily parallelized and continues until all of the failed blocks for the redundancy group are reconstructed. Additionally, this approach can be generalized to any linear erasure code; as long as the generator matrix for the code is known, the protocol remains unchanged.

4.2.2 Secure Integrity Checking

Preserving data integrity is a critical task in all long-term archives. POTSHARDS actively verifies the integrity of data using two different forms of integrity checking. The first technique requires each of the archives to periodically check its data for integrity violations using a hash stored in the header of each block on disk. The second technique is a form of inter-archive integrity checking that utilizes algebraic signatures [27] across the redundancy groups. Algebraic signatures have the prop-

erty that the signatures of the parity equals the parity of the signatures. This property is used to verify that the archives in a given redundancy group are properly storing data and are performing the required internal checks [27].

Secure, inter-archive integrity checking is achieved through algebraic signature requests over a specific interval of data. A check begins when an archive asks the members of a redundancy group for an algebraic signature over a specified interval of data. The algebraic signature forms a codeword in the erasure code used by the redundancy group and integrity over the interval of data is checked by comparing the parity of the data signatures to the signature of the parity. If the comparison check fails, then the archive(s) in violation may be found as long as the number of incorrect signatures is within the error-correction capability of the code. In general, a small signature (typically 4 bytes) is computed from a few megabytes of data. This results in very little information leakage. If necessary, restrictions may be placed on algebraic signature requests to ensure that no data is exposed during the integrity check process.

4.3 User Indexes

When shards are created, the *exact* names of the shards are returned to the user along with their archive placement locations; however, these exact pointers are *not* stored in the shards themselves, so they are not available to someone attacking the archives. Typically, a user maintains this information and the relationship between shards, fragments, objects, and files in an index to allow for fast retrieval. In the general case, the user consults her index and requests specific shards from the system. This index can, in turn, be stored within POTSHARDS, resulting in an index that can be rebuilt from a users shards with no outside information.

The index for each user can be stored in POTSHARDS as a linked list of index pages with new pages inserted at the head of the list, as shown in Figure 6. Since the index pages are designed to be stored within POTSHARDS, each page is immutable. When a user submits a file to the system, a list of mappings from the file to its shards is returned. This data is recorded in a new index page, along with a list of shards corresponding to the previous head of the index list. This new page is then submitted to the system and the shard list returned is maintained as the new head of the index list. These index root-shards can be maintained by the client application or even on a physical token, such as a flash drive or smart card.

This approach of each user maintaining their own private index has three advantages. First, since each user maintains his own index, the compromise of a user index does not affect the security of other users' data. Second,

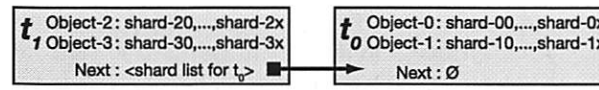


Figure 6: *User index made up of two pages. One page was created at time t_0 and the other at time t_1.*

the index for one user can be recovered with no effect on other users. Third, the system does not know about the relationship between a user's shards and their data.

In some ways, the index over a user's shards can be compared to an encryption key because it contains the information needed to rebuild a user's data. However, the user's index is different from an encryption key in two important ways. First, the user's index is not a single point of failure like an encryption key. If the index is lost or damaged, it can be recovered from the data without any input from the owner of the index. Second, full archive collusion can rebuild the index. If a user can prove a legal right to data, such as by a court subpoena, than the archives can provide all of the user's shards and allow the reconstitution of the data. If the data was encrypted, the files without the encryption key might not be accessible in a reasonable period of time.

4.4 Approximate Pointers and Recovery

Approximate pointers are used to relate shards in the same shard tuple to one another in a manner that allows recovery while still reducing an adversary's ability to launch a targeted attack. Each shard has an approximate pointer to the next shard in the fragment, with the last shard pointing back to the first and completing the cycle, as shown in Figure 3. This allows a user to recover data from their shards even if all other outside information, such as the index, is lost.

There are two ways that approximate pointers can be implemented: randomly picking a value within $R/2$ above or below the next shard's identifier, or masking off the low-order r bits ($R = 2^r$) of the next shard's identifier, hiding the true value. Currently, POTSHARDS uses the latter approach; we are investigating the tradeoffs between the two approaches. One benefit to using the $R/2$ approach is that it allows a finer-grained level of adjustment compared to the relatively coarse-grained bitmask approach.

The use of approximate pointers provides a great deal of security by preventing an intruder who compromises an archive or an inside attacker from knowing exactly which shards to steal from other archives. An intruder would have to steal *all* of the shards an approximate pointer could refer to, and would have to steal all of the shards they refer to, and so on. All of this would have to bypass the authentication mechanisms of each archive, and archives would be able to identify the access pattern of a thief, who would be attempting to obtains shards that

may not exist. Since partially reconstituted fragments cannot be verified, the intruder might have to steal *all* of the potential shards to ensure that he was able to reconstitute the fragment. For example, if an approximate pointer points to R shards and a fragment is split using (m,n) secret splitting, an intruder would have to steal, on average, $R^{m-1}/2$ shards to decode the fragment.

In contrast to a malicious user, a legitimate user with access to all of his shards can easily rebuild the fragments and, from them, the objects and files they comprise. Suppose this user created shards from fragments using an (m,n) secret splitting algorithm. A user would start by obtaining all of her shards which, in the case of recoveries, might require additional authentication steps. Once she obtains all of her shards from the archives, there are two approaches to regenerating those fragments. First, she could try every possible chain of length m, rebuilding the fragment and attempting to verify it. Second, she could narrow the list of possible chains by only attempting to verify chains of length n that represented cycles, an approach we call the *ring heuristic*. As Figure 2 illustrates, fragments include a hash that is used to confirm successful reconstitution. Fragments also include the identifier for the object from which they are derived, making the combination of fragments into objects a straightforward process.

Because the Shamir secret splitting algorithm is computationally expensive, even when combining shards that do not generate valid fragments, we use the ring heuristic to reduce the number of failed reconstitution attempts in two ways. First, the number of cycles of length n is lower than the number of paths of length m since many paths of length n do not make cycles. Second, reconstitution using the Shamir secret splitting algorithm requires that the shares be properly ordered and positioned within the share list. Though the shard ID provides a natural ordering for shards, it does not assist with positioning. For example, suppose the shards were produced with a 3 of 5 split. A chain of three shards, $\langle s_1, s_2, s_3 \rangle$, would potentially need to be submitted to the secret splitting algorithm three times to test each possible order: $\langle s_1, s_2, s_3, \phi, \phi \rangle$, $\langle \phi, s_1, s_2, s_3, \phi \rangle$, and $\langle \phi, \phi, s_1, s_2, s_3 \rangle$.

5 Experimental Evaluation

Our experiments using the current implementation of POTSHARDS were designed to measure several things. First, we wanted to evaluate the performance of the system and identify any bottlenecks. Next, we compared the behavior of the system in an environment with heavy contention for processing and network resources against that in a dedicated, lightly loaded environment. Finally, we evaluated POTSHARDS' ability to recover from the loss of an archive as well as the loss of a user index.

During our experiments, the data transformation component was run from the client's system using object sizes of 750 KB. The first layer of secret splitting used an XOR based algorithm and produced two fragments per object, and the second layer utilized a (2,3) Shamir threshold scheme. The workloads contained a mixture of PDF, Postscript files, and images. These files are representative of the content that a long-term archive might contain, although it is important to note that POTSHARDS sees all objects as the same regardless the objects' origin or content. File sizes ranged from about half a megabyte to several megabytes in size; thus, most were ingested and extracted as multiple objects.

For the local experiments, all systems were located on the same 1 Gbps network with little outside contention for computing or network resources. The client computers were equipped with two 2.74 GHz Pentium 4 processors, 2 GB of RAM and Linux version 2.6.9-22.01.1. Each of the sixteen archives were equipped with two 2.74 GHz Pentium 4 processors, 3 GB of RAM, 7.3 GB of available local hard drive space and Linux version 2.6.9-34. In contrast to the local experiments, the global-scale experiments were conducted using PlanetLab [21], resulting in considerable contention for shared resources. For these experiments, both the clients and archives were run in a slice that contained twelve PlanetLab nodes (eight archives and four clients) distributed across the globe.

The POTSHARDS prototype system itself consists of roughly 15,000 lines of Java 5.0 code. Communications between layers used Java sockets over standard TCP/IP, and the archives used Sleepycat Software's BerkeleyDB version 3.0 for persistent storage of shards.

5.1 Read and Write Performance

Our first set of experiments evaluated the performance of ingestion and extraction on a dedicated set of systems and on PlanetLab. Table 2 profiles the ingestion and extraction of one block of data. comparing the time taken on an unloaded local cluster of machines and the heavily loaded, global scale PlanetLab. In addition to the time, the table details the number of messages exchanged during the request.

As Table 2 shows, most of the time on the local cluster is spent in the transformation layer. This is to be expected as Shamir secret-splitting algorithm is compute-intensive. While slower than many encryption algorithms, such secret-splitting algorithms do not suffer from the problems discussed earlier with long-term encryption and are fast enough for archival storage. The compute-intensive nature of secret-splitting is further highlighted in the local experiments due to the local cluster's dedicated network with almost no outside cross-

Ingestion Profile		Cluster	PlanetLab
Secret Splitting	time (ms)	1509	2276
Layers	msgs in	1	1
Request	msgs out	1	1
Placement	time (ms)	37	30606
Layer	msgs in	1	1
Request	msgs out	6	6
Archive	time (ms)	67	39109
Layer	msgs in	6	6
Request	msgs out	6	6
Response Trip	time (ms)	88	54271
Total Round Trip	time (ms)	1731	95952

Extraction Profile		Cluster	PlanetLab
Request Trip	time (ms)	28	6493
Shard	time (ms)	832	29666
Acquisition	msgs	34	34
Transformation	time (ms)	1009	1698
Layer	msgs in	1	1
Response	msgs out	1	1
Total Round Trip	time (ms)	1843	31410

Table 2: *Profile of the ingestion and extraction of one object, comparing trials run on a lightly-loaded local cluster with the global-scale PlanetLab. Results are the average of 3 runs of 36 blocks per run using a* $(2,2)$ *XOR split to generate fragments and a* $(2,3)$ *Shamir split to generate shards.*

traffic. The transformation time for ingestion is greater than for extraction for two reasons. First, during ingestion, the transformation must generate many random values. Second, during extraction, the transformation layer performs linear interpolation using only those shards that are necessary. That is, given an (m,n) secret split, all n are retrieved but calculation is only done on the first m shards; the minimum required to rebuild the data. During extraction, the speed improvements in the transformation layer are balanced by the time required to collect the requested shards from the archive layer.

In a congested, heavily loaded system, the time to move data through the system begins to dominate the transformation time as the PlanetLab performance figures in Table 2 show. This is evident in the comparable times spent in the transformation layers in the two environments contrasted with the very divergent times spent on requests and responses in the two environments. For example, the extraction request trip took only 28 ms on the local cluster but required about 6.5 seconds on the PlanetLab trials. Since request messages are quite small, the difference is even more dramatic in the shard acquisition times for extraction. Here, moving the shards from the archives to the transformation layer took only 832 ms on the local cluster but over 29.5 seconds on PlanetLab.

The measurements per object represent two distinct scenarios. The cluster numbers are from a lightly-loaded,

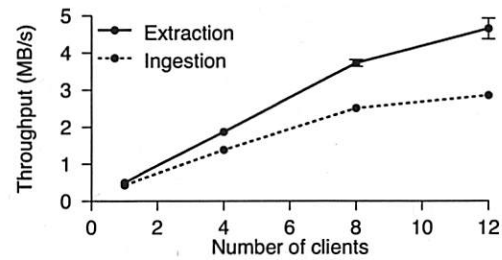
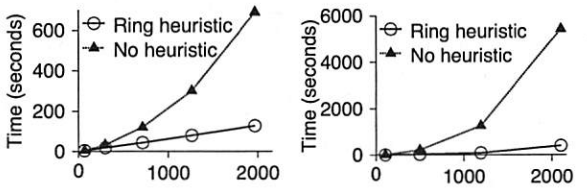


Figure 7: *System throughput with sixteen archives and a workload of 100 MB per client using the same system parameters as in Table 2.*

well-equipped and homogeneous network with unsaturated communication channels. In contrast, the PlanetLab numbers feature far more congestion and resource demands as POTSHARDS contended with other processes for both host and network facilities. However, in archival storage, latency is not as important as throughput. Thus, while these times are not adequate for low-latency applications, they are acceptable for archival storage.

The results from local tests show a per client throughput of 0.50 MB/s extraction and 0.43 MB/s ingestion—per-client performance is largely limited by the current design of the data transformation layer. In the current version, both XOR splitting and linear interpolation splitting is performed in a Java-based implementation; future versions will use $GF(2^{16})$ arithmetic in an optimized C based library. Additionally, clients currently submit objects to the data transformation component and synchronously await a response from the system before submitting the next object. In contrast, the remainder of the system is highly asynchronous. The high level of parallelism in the lower layer is demonstrated in the throughput as the number of clients increases. As Figure 7 shows, the read and write throughput scales as the number of clients increases. With a low number of clients, much of the system's time is spent waiting for a request from the secret splitting layers. As the number of clients increases, however, the system is able to take advantage of the increased aggregate requests of the clients to achieve system throughput of 4.66 MB/s for extraction and 2.86 MB/s for ingestion. Write performance is further improved through the use of asynchronous parity updates. While an ingestion response waits for the archive to write the data before being sent, it does not need to wait for the parity updates.

An additional factor to consider in measuring throughput is the storage blow-up introduced by the two levels of secret splitting. Using parameters of $(2,2)$ XOR splitting and $(2,3)$ shard splitting requires six bytes to be stored for every byte of user data. In our experiments, system throughput is measured from the client perspective even though demands inside the system are six times those



(a) 2 of 3 split. (b) 3 of 5 split.

Figure 8: *Brute force recovery time for an increasing number of shards generated using different secret splitting parameters.*

Name Space	Shards	False Rings	Time
16 bits	4190	24451	6715 sec
32 bits	4190	0	225 sec

Table 3: *Recovery time in a name space with 5447 allocated names for two different name space sizes. For larger systems, this time increases approximately linearly with system size; name density and secret splitting parameters determine the slope of the line.*

seen by the client. Nonetheless, one goal for future work is to improve system throughput by implementing asynchronous communication in the client.

5.2 User Data Recovery

In the event that the index over a user's shards is lost or damaged, user data, including the index, if it was stored in POTSHARDS, can be recovered from the shards themselves. To begin the procedure, the user authenticates herself to each of the individual archives and obtains all of her shards. The user then applies the algorithm described in Section 4.4 to rebuild the fragments and the objects that make up her data.

We ran experiments to measure the speed of the recovery process for both algorithm options. While the recovery process is not fast enough to use as the sole extraction method, it is fast enough for use as a recovery tool. Figure 8 shows the recovery times for two different secret splitting parameters. Using the ring heuristic provides a near-linear recovery time as the number of shards increases, and is much faster than the naïve approach. In contrast, recovery without using the ring heuristic results in an exponential growth. This is very apparent in Figure 8(b), which must potentially try each path three times. The ring heuristic provides an additional layer of security because a user that can properly authenticate to all of the archives and acquire all of their shards can recover their data very quickly. In contrast, an intruder that cannot acquire all of the needed shards must search in exponential time.

The density of the name space has a large effect on the time required to recover the shards. As shown in Table 3, a sparse name space results in fewer false shard rings (none in this experiment) and is almost 30 times faster than a densely packed name space. An area of future research is to design name allocation policies that balance the recovery times with the security of the shards. One simple option would be to utilize a sliding window into the name space from which names are drawn. As the current window becomes saturated it moves within the name space. This would ensure adequate density for both new names and existing names.

5.3 Archive Reconstruction

The archive recovery mechanisms were run on our local system using eight 1.5 GB archives. Each redundancy group in the experiment contained eight archives encoded using RAID 5. A 25 MB client workload was ingested into the system using (2,2) XOR splitting and (2,3) Shamir splitting, resulting in 150 MB of client shards, excluding the appropriate parity. After the workload was ingested, an archive was failed. We then used a static recovery manager that sent reconstruction requests to all of the available archives and waited for successful responses from a fail-over archive. Once the procedure completed, the contents of the failed archive and the reconstructed archive were compared. This procedure was run three times, recovering at 14.5 MB/s, with the verification proving successful on each trial. The procedure was also run with faults injected into the recovery process to ensure that the verification process was correct.

6 Discussion

While we have designed and implemented an infrastructure that supports secure long-term archival storage without the use of encryption, there are still some outstanding issues. POTSHARDS assumes that individual archives are relatively reliable; however, automated maintenance of large-scale archival storage remains challenging [3]. We plan to explore the construction of archives from autonomous power-managed disk arrays as an alternative to tape [8]. The goal would be devices that can distribute and replicate storage amongst themselves, reducing the level of human intervention to replacing disks when sufficiently many have failed.

A secure, archival system must deal with the often conflicting requirements of maintaining the secrecy of data while also providing a degree of redundancy. To this end, further work will explore the contention between these two demands in such areas as parity building. In future versions, we hope to improve the security of parity updates in which sensitive data must be passed between archives.

Currently, POTSHARDS depends on strong authentication and intrusion detection to keep data safe, but it is not clear how to defend against intrusions that may occur

over many years, even if such attacks are detected. We are exploring approaches that can refactor the data [35] so that partial progress in an intrusion can be erased by making new shards "incompatible" with old shards. Unlike the failure of an encryption algorithm, which would necessitate wholesale re-encryption, refactoring for security could be done over time to limit the window over which a slow attack could succeed. Refactoring could also be applicable to secure migration of data to new storage devices.

We have introduced the approximate pointer mechanism as a means of making data recovery more tractable while maintaining security. While we believe they are useful in this capacity, we admit that there is more work to be done in understanding their nature. Specifically, we plan on exploring the relationship between the ID namespace and approximate pointer parameters.

We would also like to reduce the storage overhead in POTSHARDS, and are considering several approaches to do so. Some information dispersal algorithms may have lower overheads than Shamir secret splitting; we plan to explore their use, assuming that they maintain the information-theoretic security provided by our current algorithm.

The research in POTSHARDS is only concerned with preserving the bits that make up files; understanding the bits is an orthogonal problem that must also be solved. Others have begun to address this problem [9], but maintaining the semantic meanings of bits over decades-long periods may prove to be an even more difficult problem than securely maintaining the bits themselves.

7 Conclusions

This paper introduced POTSHARDS, a system designed to provide secure long-term archival storage to address the new challenges and new security threats posed by archives that must securely preserve data for decades or longer.

In developing POTSHARDS, we made several key contributions to secure long-term data archival. First, we use multiple layers of secret splitting, approximate pointers, and archives located in independent authorization domains to ensure secrecy, shifting security of long-lived data away from a reliance on encryption. The combination of secret splitting and approximate pointers forces an attacker to steal an exponential number of shares in order to reconstitute a single fragment of user data; because he does not know which particular shares are needed, he must obtain *all* of the possibly-required shares. Second, we demonstrated that a user's data can be rebuilt in a relatively short time from the stored shares *only* if sufficiently many pieces can be acquired. Even a sizable (but incomplete) fraction of the stored pieces from a subset of the archives will not leak information, ensuring that data stored in POTSHARDS will remain secret. Third, we made intrusion detection easier by dramatically increasing the amount of information that an attacker would have to steal and requiring a relatively unusual access pattern to mount the attack. Fourth, we ensure long-term data integrity through the use of RAID algorithms across multiple archives, allowing POTSHARDS to utilize heterogeneous storage systems with the ability to recover from failed or defunct archives and a facility to migrate data to newer storage devices.

Our experiments show that the current prototype implementation can store user data at nearly 3 MB/s and retrieve user data at 5 MB/s. Since POTSHARDS is an archival storage system, throughput is more of a concern than latency, and these throughputs exceed typical long-term data creation rates for most environments. Since the storage process is parallelizable, additional clients increase throughput until the archives' maximum throughput is reached; similarly, additional archives linearly increase maximum system throughput.

By addressing the long-term threats to archival data while providing reasonable performance, POTSHARDS provides reliable data protection specifically designed for the unique challenges of secure archival storage. Storing data in POTSHARDS ensures not only that it will remain available for decades to come, but also that it will remain secure and can be recovered by authorized users even if all indexing is lost.

Acknowledgments

We would like to thank our colleagues in the Storage Systems Research Center (SSRC) who provided valuable feedback on the ideas in this paper. This research was supported by the Petascale Data Storage Institute, UCSC/LANL Institute for Scalable Scientific Data Management and by SSRC sponsors including Los Alamos National Lab, Livermore National Lab, Sandia National Lab, Digisense, Hewlett-Packard Laboratories, IBM Research, Intel, LSI Logic, Microsoft Research, Network Appliance, Seagate, Symantec, and Yahoo.

References

[1] Health Information Portability and Accountability act, Oct. 1996.

[2] ADYA, A., BOLOSKY, W. J., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, Dec. 2002), USENIX.

[3] BAKER, M., SHAH, M., ROSENTHAL, D. S. H., ROUSSOPOULOS, M., MANIATIS, P., GIULI, T., AND BUNGALE, P. A fresh

look at the reliability of long-term digital storage. In *Proceedings of EuroSys 2006* (Apr. 2006), pp. 221–234.

[4] CHANG, F., JI, M., LEUNG, S.-T. A., MACCORMICK, J., PERL, S. E., AND ZHANG, L. Myriad: Cost-effective disaster tolerance. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (San Francisco, CA, Jan. 2002).

[5] CHOI, S. J., YOUN, H. Y., AND LEE, B. K. An efficient dispersal and encryption scheme for secure distributed information storage. *Lecture Notes in Computer Science 2660* (Jan. 2003), 958–967.

[6] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science 2009* (2001), 46+.

[7] CLEVERSAFE. Highly secure, highly reliable, open source storage solution. Available from http://www.cleversafe.org/, June 2006.

[8] COLARELLI, D., AND GRUNWALD, D. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)* (Nov. 2002).

[9] GLADNEY, H. M., AND LORIE, R. A. Trustworthy 100-year digital objects: Durable encoding for when it's too late to ask. *ACM Transactions on Information Systems 23*, 3 (July 2005), 299–324.

[10] GOLDBERG, A. V., AND YIANILOS, P. N. Towards an archival intermemory. In *Advances in Digital Libraries ADL'98* (April 1998), pp. 1–9.

[11] GOODSON, G. R., WYLIE, J. J., GANGER, G. R., AND REITER, M. K. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the 2004 Int'l Conference on Dependable Systems and Networking (DSN 2004)* (June 2004).

[12] GUNAWI, H. S., AGRAWAL, N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SCHINDLER, J. Deconstructing commodity storage clusters. In *Proceedings of the 32nd Int'l Symposium on Computer Architecture* (June 2005), pp. 60–71.

[13] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)* (May 2005).

[14] HAND, S., AND ROSCOE, T. Mnemosyne: Peer-to-peer steganographic storage. *Lecture Notes in Computer Science 2429* (2002), 130–140.

[15] IYENGAR, A., CAHN, R., GARAY, J. A., AND JUTLA, C. Design and implementation of a secure distributed data repository. In *Proceedings of the 14th IFIP International Information Security Conference (SEC '98)* (Sept. 1998), pp. 123–135.

[16] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: scalable secure file sharing on untrusted storage. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)* (San Francisco, CA, Mar. 2003), USENIX, pp. 29–42.

[17] KEETON, K., SANTOS, C., BEYER, D., CHASE, J., AND WILKES, J. Designing for disasters. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST)* (San Francisco, CA, Apr. 2004).

[18] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H., AND BAKER, M. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems 23*, 1 (2005), 2–50.

[19] MILLER, E. L., LONG, D. D. E., FREEMAN, W. E., AND REED, B. C. Strong security for network-attached storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (Monterey, CA, Jan. 2002), pp. 1–13.

[20] OXLEY, M. G. (H.R.3763) Sarbanes-Oxley Act of 2002, Feb. 2002.

[21] PETERSON, L., MUIR, S., ROSCOE, T., AND KLINGAMAN, A. PlanetLab Architecture: An Overview. Tech. Rep. PDN–06–031, PlanetLab Consortium, May 2006.

[22] PLANK, J. S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software—Practice and Experience (SPE) 27*, 9 (Sept. 1997), 995–1012. Correction in James S. Plank and Ying Ding, Technical Report UT-CS-03-504, U Tennessee, 2003.

[23] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (Monterey, California, USA, 2002), USENIX, pp. 89–101.

[24] RABIN, M. O. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36* (1989), 335–348.

[25] RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. Pond: the OceanStore prototype. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)* (Mar. 2003), pp. 1–14.

[26] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)* (Dec. 1999), pp. 110–123.

[27] SCHWARZ, S. J., T., AND MILLER, E. L. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS '06)* (Lisboa, Portugal, July 2006), IEEE.

[28] SHAMIR, A. How to share a secret. *Communications of the ACM 22*, 11 (Nov. 1979), 612–613.

[29] STINSON, D. R. *Cryptography Theory and Practice*, 2nd ed. Chapman & Hall/CRC, Boca Raton, FL, 2002.

[30] STONEBRAKER, M., AND SCHLOSS, G. A. Distributed RAID—a new multiple copy algorithm. In *Proceedings of the 6th International Conference on Data Engineering (ICDE '90)* (Feb. 1990), pp. 430–437.

[31] STORER, M., GREENAN, K., MILLER, E. L., AND MALTZAHN, C. POTSHARDS: Storing data for the long-term without encryption. In *Proceedings of the 3rd International IEEE Security in Storage Workshop* (Dec. 2005).

[32] STORER, M. W., GREENAN, K. M., AND MILLER, E. L. Long-term threats to secure archives. In *Proceedings of the 2006 ACM Workshop on Storage Security and Survivability* (Oct. 2006).

[33] SUBBIAH, A., AND BLOUGH, D. M. An approach for fault tolerant and secure data storage in collaborative work environements. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability* (Fairfax, VA, Nov. 2005), pp. 84–93.

[34] WALDMAN, M., RUBIN, A. D., AND CRANOR, L. F. Publius: A robust, tamper-evident, censorship-resistant web publishing system. In *Proceedings of the 9th USENIX Security Symposium* (Aug. 2000).

[35] WONG, T. M., WANG, C., AND WING, J. M. Verifiable secret redistribution for threshold sharing schemes. Tech. Rep. CMU-CS-02-114-R, Carnegie Mellon University, Oct. 2002.

[36] WYLIE, J. J., BIGRIGG, M. W., STRUNK, J. D., GANGER, G. R., KILIÇÇÖTE, H., AND KHOSLA, P. K. Survivable storage systems. *IEEE Computer* (Aug. 2000), 61–68.

[37] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)* (Tokyo, Japan, Apr. 2005), IEEE.

Dandelion: Cooperative Content Distribution with Robust Incentives

Michael Sirivianos Jong Han Park Xiaowei Yang Stanislaw Jarecki
Department of Computer Science
University of California, Irvine
{*msirivia,jonghanp,xwy,stasio*}*@ics.uci.edu*

Abstract

Content distribution via the Internet is becoming increasingly popular. To be cost-effective, commercial content providers are considering the use of peer-to-peer (P2P) protocols such as BitTorrent to save on bandwidth costs and to handle peak demands. However, when an online content provider uses a P2P protocol, it faces a crucial issue: how to incentivize its clients to upload to their peers.

This paper presents Dandelion, a system designed to address this issue in the case of paid content distribution. Unlike previous solutions, most notably BitTorrent, Dandelion provides robust (provably non-manipulable) incentives for clients to upload to others. In addition, unlike systems with tit-for-tat-based incentives, a client is motivated to upload to its peers even if the peers do not have content that interests the client. A client that honestly uploads to its peers is rewarded with credit, which can be redeemed for various types of rewards, such as discounts on paid content.

In designing Dandelion, we trade scalability for the ability to provide robust incentives. The evaluation of our prototype system on PlanetLab demonstrates the viability of our approach. A Dandelion server that runs on commodity hardware with a moderate access link is capable of supporting up to a few thousand clients. These clients can download content at rates comparable to those of BitTorrent clients.

1 Introduction

Content distribution via the Internet is becoming increasingly popular among the entertainment industry and the consumers alike. A survey showed that Apple's iTunes music store sold more music than Tower Records and Borders in the US in the summer of 2005 [10]. A number of key content producers, such as Universal, are now launching download to own services [15]. However, the increasing demand for digital content is overwhelming the infrastructure of online content providers [13].

An attractive approach for commercial online content distribution is the use of peer-to-peer (P2P) protocols. This approach does not require a content provider to overprovision its bandwidth to handle peak demands, nor does it require the provider to purchase service from a third-party such as Akamai. Instead, a P2P protocol such as BitTorrent [26] harnesses its clients' bandwidth for file distribution, and saves the bandwidth and computing resources of a content provider. Leading content providers such as Warner Bros [16] and 20th Century Fox [11] have now partnered with BitTorrent, Inc. EMI [12] has announced a plan to launch a P2P music distribution service. This recent trend indicates that P2P protocols enable a site to cost-effectively distribute content.

When an online content provider uses a peer-to-peer protocol, it faces a crucial issue: how to motivate clients that possess content to upload to others. This issue is of paramount importance because the performance of a P2P network is highly dependent on the users' willingness to contribute their uplink bandwidth. However, selfish (rational) users tend not to share their bandwidth without external incentives [36]. Although the popular BitTorrent protocol has incorporated the rate-based tit-for-tat incentive mechanism for users to upload static content, this mechanism bears two weaknesses. First, it does not encourage clients to seed, i.e. to upload to other peers after completing the file download. Second, it is vulnerable to manipulation [37, 44, 45, 49, 50], allowing modified clients to free-ride and still achieve a better downloading rate than compliant clients (Section 2.2).

The purpose of this work is to explore the design space of a P2P content distribution protocol that addresses this issue. We present the design and implementation of Dandelion, a cooperative paid content distribution protocol that uses non-manipulable virtual-currency-based incentives to encourage uploading and to address free-riding.

Our protocol guarantees strict fair exchange of content uploads for virtual currency (credit). A client cannot download content from *selfish* peers (i.e. peers that do

not upload unless they expect to be rewarded) without paying credit, neither it can obtain credit for uploads it did not perform. This protocol property provides robust incentives for selfish peers to contribute their bandwidth in the following two ways. First, credit can be redeemed at a content provider for a discount on the content, or for other types of monetary awards. Given appropriate pricing schemes, a selfish client is motivated to serve content to its peers regardless of whether its peers possess content that interests it. Second, the protocol prevents free-riding, because, provably, the only way a client can obtain valid content from selfish peers or can earn credit is by paying credit or uploading valid content, respectively. Hence, we believe that our protocol can increase the aggregate upload bandwidth of a P2P content distribution system and improve downloading times.

The use of virtual currency for incentives has been proposed in several P2P content distribution systems [3, 7, 8, 17, 29, 52, 53], but a key challenge, how to make the virtual-currency-based system efficient and practical while robust to manipulation, is left unaddressed (Sections 2.1 and 2.4). We address this challenge based on the insight that in the problem domain of online content distribution, the content provider itself is a trusted third party and can mediate the content exchange between its clients. Based on this observation, we design a protocol in which clients exchange data for credit and the server mediates this exchange. The server uses only efficient symmetric cryptography on critical data paths and sends only short messages to its clients.

Our work makes the following contributions:

1) An efficient cryptographic fair exchange scheme for trading data uploads for virtual currency, which is suitable for P2P content distribution. Our scheme is based on symmetric key cryptography, and is provably robust to client cheating. A client that does not upload or uploads garbage to its peers cannot claim credit. A client cannot download correct content from selfish peers without the client being charged and the peers rewarded.

2) The design and implementation of Dandelion. To the best of our knowledge, Dandelion is the first implemented P2P static content distribution protocol that uses symmetric cryptography in order to provide robust incentives for clients to upload paid content to their peers. Our system's evaluation on PlanetLab [24] identifies the scalability limits of our incentive mechanism and demonstrates the plausibility of our approach.

In this paper, we use a BitTorrent-like terminology. A *seeder* refers to a client that uploads to its peers despite not being interested in the content being distributed (e.g. a client that uploads although it has completed its file download). A *leecher* refers to a client that is interested in the content being distributed (e.g. a client that has not completed its file download). A *free-rider* refers to a client that only downloads from others but does not upload. A *swarm* refers to all clients that actively participate in the protocol for a given content item.

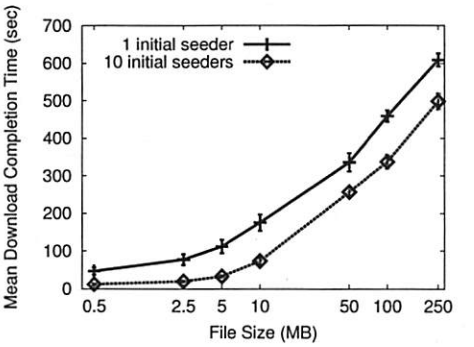


Figure 1: **Mean download completion times of ~200 CTorrent 1.3.4 leechers as a function of file size, for 1 initial seeder and 10 initial seeders. The mean file download completion times are extracted over 10 runs. Error bars correspond to 95% confidence intervals.**

The rest of this paper is organized as follows. Section 2 describes existing incentive mechanisms in P2P protocols, and cryptographic fair exchange schemes. Section 3 describes the design of Dandelion. Section 4 analyzes the security of our design. Sections 5 and 6 present our system's implementation and experimental evaluation, respectively. We conclude in Section 7.

2 Background and Related Work

In this section we motivate the design of Dandelion by describing existing P2P content distribution incentive mechanisms and their weaknesses. In addition, we discuss previous work on cryptographic fair exchange.

2.1 Impact of Seeding

The popular BitTorrent protocol employs the rate-based tit-for-tat incentive mechanism [26]. A client unchokes (i.e. uploads to) at most four to ten clients for a given file, in parallel. Most of the unchoked peers are the peers that upload useful parts of the file to the client at the fastest rates and are interested in the client's content. The client also *optimistically* unchokes one or two peers that are not among the fastest uploaders, in expectation of future reciprocation. The list of unchoked peers is typically revised every 10 seconds.

This mechanism mitigates free-riding but does not provide explicit incentives for seeding. Although several BitTorrent deployments rely on clients to honestly report their uploading history [17], and use this history to decide which clients can join a swarm, practice has shown that clients can fake their upload history reports [4].

In contrast, Dandelion's non-manipulable and centrally maintained virtual currency enables a content distributor to reliably keep record of the amount of content a selfish client has uploaded to its peers. The distributor can use this record to provide robust incentives for a selfish client to upload to its peers regardless of whether the peers have content that interests the client.

Seeders can substantially improve download completion times, because they increase the file availability and the aggregate upload bandwidth. Figure 1 shows the impact of seeders. We run two BitTorrent experiments on PlanetLab, with one and ten initial seeders, respectively. Initial seeders are clients that have the complete file prior to the start of the distribution. In each experiment, we run ~200 CTorrent 1.3.4 [2] leechers on distinct PlanetLab nodes to simultaneously download a file. Upon completion of their download, leechers remain online seeding the file. As can be seen, the mean file download completion time decreases considerably when there are ten initial seeders, especially for small files of a few MB.

2.2 Free-riding in BitTorrent

A general observation is that since BitTorrent's tit-for-tat incentives reward cooperative clients with improved download times, clients are always incented to upload. Therefore, free-riding should not be an issue in BitTorrent networks. This observation relies on the assumption that users aim only at maximizing their download rates. However, in practice, several BitTorrent users can be reluctant to upload even if uploading improves their download times. For example, users with access providers that impose quotas on outgoing traffic or users with limited uplink bandwidth (e.g. 1.5Mbps/128Kbps ADSL) may wish to save their uplink for other critical tasks.

Considering the tradeoff between performance and susceptibility to free-riding [31], BitTorrent purposely does not implement a strict tit-for-tat (TFT) strategy. In particular, it employs rate-based instead of chunk-level TFT, and BitTorrent clients optimistically unchoke peers for relatively long periods of time (10 to 30 seconds). Furthemore, BitTorrent seeders select peers to upload to regardless of whether those peers upload to others.

Based on the above observations and previous work on BitTorrent exploitation [37, 44, 49], in [50], we modify a CTorrent-1.3.4 client to employ the "large view" exploit to free-ride. The client obtains a larger than normal view of the swarm, either by repeatedly requesting partial views from the BitTorrent tracker or by exchanging views with its peers [6, 9]. Subsequently, it connects to all peers in its view, while it does not upload any content. Using this exploit in a sufficiently large swarm, a modified client can substantially increase the frequency with which it becomes optimistically unchoked, comparing to a compliant client, which typically connects to 50-100 peers. It can also find more seeders, which do not employ tit-for-tat.

In particular, we show that our modified free-rider client is able to download faster than its tit-for-tat compliant counterpart in 12 out of 15 randomly selected public torrents, for file sizes between 500MB to 2 GB and swarm sizes of 50 to 1000 peers. We also experiment with PlanetLab residing swarms that comprise of ~300 leechers that are rate-limited at 30KB/sec and one initial seeder that is rate-limited at 120KB/sec. When compliant clients comprise 90% of the PlanetLab-residing swarm, free-riders download faster than compliant clients in their swarm and slightly worse than compliant clients in a swarm with no free-riders.

The same weakness of BitTorrent's incentives is experimentally demonstrated in a recent work by Locher et al. [45], which was almost concurrent with ours.

Drawing from the above observations, we believe that the "large view" exploit has the potential to be widely adopted and could lead to system-wide performance degradation in BitTorrent swarms. Dandelion explicitly addresses this issue, because its provably non-manipulable incentives enable a content distributor to reliably track the amount of content a client has downloaded from selfish peers, and charge the client accordingly.

2.3 Pairwise Currency as Incentives

In P2P content distribution protocols that employ pairwise virtual currency (credit) for incentives, clients maintain distinct credit balances for each of their peers. In this context, credit refers to any metric of a peer's cooperativeness.

An eMule [7] client rewards cooperative peers by reducing the time the peers have to wait until they are served by the client. Swift [52] introduces a pairwise credit-based trading mechanism for peer-to-peer file sharing networks and examines the available peer strategies. In [37], the authors suggest tackling free-riding in BitTorrent by employing chunk-level tit-for-tat, which is similar to pairwise credit incentives. These pairwise credit-based incentive mechanisms bear weaknesses that are similar to the ones of rate-based tit-for-tat: a) they provide no explicit incentives for seeding; and b) they can be manipulated by free-riders that obtain a "large view" of the network, and initiate short-lived sessions with numerous peers to exploit the initial offers in pairwise transactions.

Scrivener [29] combines pairwise credit balances with a transitive trading mechanism, which is based on a flavor of distributed reputation. MNet [8] uses a combination of pairwise balances and tokens that can be cashed in a central broker. When the debt during pairwise transactions exceeds a specified threshold, the side with the

negative balance transfers a credit token to the other by contacting a broker. Since both Scrivener and MNet do not provide strong fair exchange guarantees of content uploads for credit, they can be manipulated in a way similar to the "large view" exploit.

Keidar et al. [38] present the design of a P2P multicast protocol, which is formally proven to enforce cooperation among selfish leechers. To prove cooperation, the authors assumed that selfish leechers abide by a predetermined strategy, which specifies how many peers a leecher can have. However, the recent work on BitTorrent exploitation [45, 50], which has partly motivated our system's design, has demonstrated that this assumption may be too restrictive.

BAR Gossip [42] is suitable for P2P streaming of live content. Owing to its cryptographic fair exchange mechanism and its verifiable peer selection, the system is robust to clients that attempt to free-ride by obtaining a "large view." However, its verifiable peer selection technique assumes that no client can join the network after the streaming session starts. Since BAR Gossip is designed for P2P streaming, it does not need to provide incentives for seeding. Therefore, it ensures the fair exchange of content uploads between clients that are interested in the same live broadcast. On the other hand, Dandelion, which needs to incent seeding for static content distribution or video on demand, guarantees fair exchange of content uploads for virtual currency.

2.4 Global Currency as Incentives

It has been widely proposed to use global virtual currency to provide incentives in P2P content distribution systems. This is the basis of the incentive mechanism employed by Dandelion: for each client, the system maintains a credit balance, which is used to track the bandwidth that the client has contributed to the network.

Karma [53] employs a global credit bank and certified-mail-based [48] fair exchange of content for reception proofs. It distributes credit management among multiple nodes. Karma's distributed credit management improves scalability. However, it does not guarantee the integrity of the global currency when the majority of the nodes that comprise the distributed credit bank are malicious or in a highly dynamic network. In contrast, Dandelion's centrally maintained global currency is non-manipulable by clients. Thus, it enables a server to offer monetary rewards based on client credit balances, providing strong incentives for clients to cooperate.

Horne et al. [35] proposed an encryption- and erasure-code-based fair exchange scheme for exchange of content for proofs of service, but did not provide an experimental evaluation. Their scheme detects cheating with probabilistic guarantees, whereas Dandelion deterministically detects and punishes cheaters.

Li et al. [43] proposed a scheme for incentives in P2P environments that uses fair exchange of proof of service with chunks of content. The selfish client encrypts a chunk and sends it to its peer, the peer responds with a public-key cryptographic proof of service, and the client completes the transaction by sending the decryption key. A trusted third party (TTP) is involved only in the following cases: a) the selfish client presents the proofs of service to obtain credit; b) the peer complains for receiving an invalid chunk; and c) the peer complains for not receiving the decryption key from the selfish client. However, unless the server incurs the high cost of frequently renewing the public key certificates of each client, the credit system is vulnerable to clients that obtain content from selfish peers, despite those clients not having sufficient credit. In contrast, in Dandelion, the TTP mediates every chunk exchange, effectively preventing a client from obtaining any chunks from selfish peers without having sufficient credit.

PPay [55] and WhoPay [54] are recent *micropayments* proposals that employ public key cryptography and are designed for the P2P content distribution case. These systems do not guarantee fair exchange of content for payment. Free-riders may establish short-lived sessions to many peers, download portions of content from them or obtain payments, and thereby obtain a substantial amount of content or credit without paying or uploading.

2.5 Cryptographic Fair Exchange

There are two main classes of solutions for the classic cryptographic fair exchange problem. One uses simultaneous exchange by interleaving the sending of the message with the sending of the receipt [22, 25, 27, 30, 47]. These protocols rely on the assumption of equal computational and bandwidth capacity, which does not suit the heterogeneous P2P setting.

The other class relies on the use of a trusted [18, 19, 56, 57] or semi-trusted [32, 33] third party (TTP). The main differences of our scheme are as follows: 1) In [18, 19, 57] the TTP cannot decide whether a party has misbehaved, but can only complete the transaction itself if presented with proof that the parties initially intended to perform the transaction. They assume that the cost of sending the data is small and can be repeated by the TTP. However in Dandelion, transmission of data is the most expensive resource and our scheme aims at the fair exchange of this resource; 2) Unlike [32] and [33], our scheme does not rely on untrusted clients to become semi-TTP; 3) Unlike [56], our scheme does not use public key cryptography for encryption and for committing to messages, and only requires one client rather than two to contact the TTP for each transaction. The technique they use to determine whether a message originates from a party is similar to the one used by our complaint mech-

anism, but our work also addresses the specifics of determining the validity of the message.

3 Design

In this section we describe the system model and the design of Dandelion.

3.1 Overview

Our design is based on the premise that a low cost server does not have sufficient network I/O resources to directly serve content to its clients under a flash crowd event [14]. It may however, have sufficient CPU, memory, and memory/disk/network I/O resources to execute many symmetric cryptography operations, to maintain TCP connection and protocol state for many clients, to access its client's protocol state, and to receive and send short messages [34]. However, CPU, memory and I/O are still limited resources. Therefore we aim at making the design as efficient as possible.

Under normal workload, A Dandelion server behaves similar to a web/ftp, streaming or video on demand server, i.e. it directly serves content to its clients. When a server is overloaded, it enters a *peer-serving* mode. Upon receiving a request, the server redirects the client to other clients that are able to serve the requests for content. In the peer-serving mode, a Dandelion system is reminiscent of BitTorrent, in the sense that a server splits content into verifiable *chunks*, and clients exchange carefully selected chunks. As is the case with BitTorrent, the content is split into multiple chunks in order to enable clients to upload as soon as they receive and verify a small portion of the content. It is also split in order to increase the entropy of content in the network, facilitating chunk exchanges among peers. We discuss the tradeoffs in selecting a chunk size in the case of static content distribution in Section 6.3.1.

However, our protocol uses a different incentive mechanism. The server maintains a virtual economy and associates each client with its credit balance. It entices selfish clients to upload to others by explicitly rewarding them with virtual credit, while it charges clients that download content from selfish peers.

3.2 System Model

We describe the system model under which Dandelion is designed to operate. We assume three types of clients, which we define as follows:

- **Malicious** clients aim at harming the system. They misbehave as follows: a) they may attempt to cause other clients to be blacklisted or charged for chunks they did not obtain; b) they may attempt to perform a Denial of Service (DoS) attack against the server or selected clients (this attack would involve only protocol messages, as we consider bandwidth or connection flooding attacks outside the scope of this work); and c) they may upload invalid chunks aiming at disrupting the distribution of content.
- **Selfish** (rational) clients share a utility function. This function describes the cost they incur when they upload a chunk to their peers and when they pay virtual currency to download a chunk. It also describes the benefit they gain when they are rewarded in virtual currency for correct chunks they upload and when they obtain chunks they wish to download. A selfish client aims at maximizing its utility. We assume that the content provider prices a peer's accumulated virtual currency appropriately: the benefit that a selfish client gains from acquiring virtual currency for content it uploads exceeds the cost of utilizing its uplink to upload it.

A selfish client may consider manipulating the credit system in order to maximize its utility by misbehaving as follows: a) it may not upload chunks to a peer, and yet claim credit for them; b) it may upload garbage either on purpose or due to communication failure, and yet claim credit; c) it may obtain chunks from selfish clients, and yet attempt to avoid being charged; d) it may attempt to download content from selfish peers without having sufficient credit; and e) it may attempt to boost its credit by colluding with other clients or by opening multiple Dandelion accounts.

- **Altruistic** clients upload correct content to their peers regardless of the cost they incur and they do not expect to be rewarded.

We assume weak security of the IP network, meaning that a malicious or a selfish client cannot interfere with the routing and forwarding function, and cannot corrupt messages, but it can eavesdrop messages. In addition, we assume that communication errors may occur during message transmissions.

In the rest of this section we describe the design of Dandelion, which explicitly addresses the challenges posed by selfish and malicious clients, as well as the challenges posed by the communication channel.

3.3 Credit Management

Dandelion's incentive mechanism creates a virtual economy, which enables a variety of application scenarios. A client spends $\Delta_c > 0$ credit units for each chunk it downloads from a selfish client and a selfish client earns $\Delta_r > 0$ credit units for each chunk it uploads to a client. A client can acquire a chunk only if its credit is greater than Δ_c. We set $\Delta_c = \Delta_r$, so that two colluding clients cannot increase the sum of their credit, by falsely claiming that they upload to each other.

Our protocol is intended for the case in which users maintain paid accounts with the content provider, such as in iTunes. A client is awarded sufficient initial credit to

download the complete paid content from the server. The content provider may redeem a client's credit for monetary rewards, such as discounts on content prices or service membership fees, similar to the mileage programs of airline companies. This incents a client to upload to others and earn credit. A user cannot boost its credit by presenting multiple IDs (Sybil attack [28]) and claiming to have uploaded to some of its registered IDs. This is because each user maintains an authenticated paid account with the provider. The user essentially purchases its initial credit, and the net sum in an upload-download transaction between any two IDs is zero.

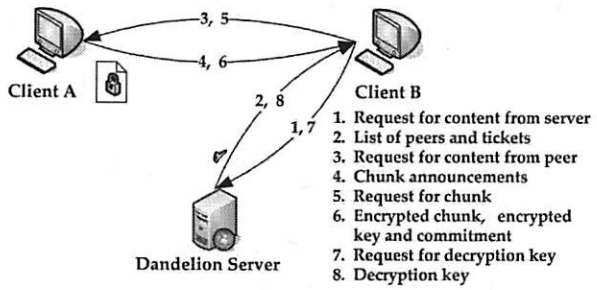


Figure 2: **The Dandelion protocol. The numbers on the arrows correspond to the listed protocol messages and the steps listed in Section 3.4.2. The messages are sent in the order they are numbered.**

3.4 Robust Incentives

This section describes Dandelion's cryptographic fair-exchange-based protocol.

3.4.1 Setting

By $\langle X \rangle$ we denote the description of an entity or object, e.g. $\langle X \rangle$ denotes a client X's Dandelion ID. K_S is S's master secret key, H is a cryptographic hash function such as SHA-1, *MAC* is a Message Authentication Code such as HMAC [20], and p refers to a time period. By p_X we denote p at client or server X.

Due to host mobility and NATs, we do not use Internet address (IP or IP/source-port) to associate credit and other persistent protocol information with clients. Instead, each user applies for a Dandelion account and is associated with a persistent ID. The server S associates each client with its authentication information (client ID and password), the content (e.g. a file) $\langle F \rangle$ it currently downloads or seeds, its credit balance, and the content it can access. The clients and the server maintain loosely synchronized clocks.

Every client A that wishes to join the network must establish a transport layer secure session with the server S, e.g. using TLS [1]. A client sends its ID and password over the secure channel. The server S generates a secret key and symmetric encryption initialization vector pair, denoted K_{SA}, which is shared with A. K_{SA} is efficiently computed as $K_{SA} = (H(K_S, \langle A \rangle, p, 0), H(K_S, \langle A \rangle, p, 1))$. K_{SA} is also sent over the secure channel. This key is used both for symmetric encryption and for computing a MAC. For MAC computation, we use only the secret key portion of K_{SA}. The rest of the messages that are exchanged between the server and the clients are sent over an insecure communication channel (e.g. plain TCP), which must originate from the same IP as the secure session. Similarly, all messages between clients are sent over an insecure communication channel.

Each client B exchanges only short messages with the server. To prevent forgery of the message source and replay attacks, and to ensure the integrity of the message, each message includes a sequence number and a digital signature. The signature is computed as the MAC of the message, keyed with the secret key K_{SB} that B shares with the server. Each time a client or the server receive a message from each other, they check whether the sequence number succeeds the sequence number of the previously received message and whether the MAC-generated signature verifies. If either of the two conditions is not satisfied, the message is discarded. The sequence number is reset when time period p changes.

The server initiates re-establishment of shared keys K_{SA} with the clients upon p change in order to: a) prevent attackers from inferring K_{SA} by examining the encrypted content and the MACs used by the protocol; and b) allow the reuse of message sequence numbers once the numbers reach a high threshold, while preventing attackers from replaying previously signed and sent messages. S tolerates some lag in the p assumed by a client.

3.4.2 Protocol Description

To provide robust incentives for cooperation under the model described in Section 3.2, Dandelion employs a cryptographic fair-exchange mechanism. Our fair-exchange protocol involves only efficient symmetric cryptographic operations. The server acts as the trusted third party (TTP) mediating the exchanges of content for credit among its clients. When a client A uploads to a client B, it sends encrypted content to client B. To decrypt, B must request the decryption key from the server. The requests for keys serve as the proof that A has uploaded some content to B. Thus, when the server receives a key request, it credits A for uploading content to B, and charges B for downloading content.

When a client A sends invalid content to B, B can determine that the content is invalid only after receiving the decryption key and being charged. To address this problem, our design includes a non-repudiable complaint mechanism. If A intentionally sends garbage to B, A cannot deny that it did. In addition, B is prevented from falsely claiming that A has sent it garbage.

The following description omits the sequence number and the signature in the messages between clients and the server. Figure 2 depicts the message flow in our protocol.

Step 1: The protocol starts with the client B sending a request for the content item $\langle F \rangle$ to S.

$B \longrightarrow S$: [content request] $\langle F \rangle$

Step 2: If B has access to $\langle F \rangle$, S chooses a short list of clients $\langle A \rangle_{\text{list}}$, which are currently in the swarm for $\langle F \rangle$. The policy with which the server selects the list $\langle A \rangle_{\text{list}}$ depends on the specifics of the content distribution system. Each list entry, besides the ID of the client, also contains the client's inbound Internet address. For every client in $\langle A \rangle_{\text{list}}$, S sends a ticket $T_{SA} = MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, t]$ to B. t is a timestamp, and $\langle A \rangle$ is a client in $\langle A \rangle_{\text{list}}$. The tickets T_{SA} are only valid for a certain amount of time T (considering clock skew between A and S) and allow B to request chunks of the content $\langle F \rangle$ from client A. When T_{SA} expires and B still wishes to download from A, it requests a new T_{SA} from S.

To ensure integrity in the case of static content or video on demand, S also sends to B the SHA-1 hash $H(c)$ for all chunks c of $\langle F \rangle$. For the case of live streaming content, the content provider augments the chunks it generates with his public key signature on their hash and ID, as $sign(H(c), \langle c \rangle)$. Clients append this signature to all the chunks they upload.

$S \longrightarrow B$:[content response] $T_{SA}, \langle A \rangle_{\text{list}}, H(c)_{\text{list}}, \langle F \rangle, t, p_S$

Step 3: The client B forwards this request to each $A \in \langle A \rangle_{\text{list}}$.

$B \longrightarrow A$:[content request] $T_{SA}, \langle F \rangle, t, p_S$

Step 4: If *current-time* $\leq ts + T$ and T_{SA} is not in A's cache, A verifies if $T_{SA} = MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, t]$. The purpose of this check is to mitigate DoS attacks against A; it allows A to filter out requests from clients that are not authorized to retrieve the content or from clients that became blacklisted. As long as B remains connected to A, it periodically renews its T_{SA} tickets by requesting them from S. If the verification fails, A drops this request. Also, if p_S is greater than A's current epoch p_A, A learns that it should renew its key with S soon. Otherwise, A caches T_{SA} and periodically sends the chunk announcement message described below, for as long as the timestamp t is fresh. This message contains a list of chunks that A owns, $\langle c \rangle_{\text{list}}$. B also does so in separate chunk announcement messages. The specifics of which chunks are announced and how frequently depend on the type of content distribution. For example, in the case of static content distribution, the initial chunk announcement would contain the IDs of all the chunks A owns, while the periodically sent announcement would contain the IDs of newly acquired chunks.

$A \longrightarrow B$:[chunk announcement] $\langle c \rangle_{\text{list}}$

Step 5: B and A determine which chunks to download from each other according to a chunk selection policy; BitTorrent's locally-rarest-first is suitable for static content dissemination, while for streaming content or video on demand other policies are appropriate [23, 42]. A can request chunks from B, after it requests and retrieves T_{SB} from S. B sends a request for the missing chunk c to A.

$B \longrightarrow A$:[chunk request] $T_{SA}, \langle F \rangle, \langle c \rangle, t, p_S$

Step 6: B's chunk requests are served by A as long as the timestamp t is fresh, and T_{SA} is cached or T_{SA} verifies. If A is altruistic, it sends the chunk c to B in plaintext and the per-chunk transaction ends here. Otherwise, A encrypts c using a symmetric encryption algorithm Enc, as $C = Enc_{k_{\langle c \rangle}}(c)$. $k_{\langle c \rangle}$ is a random secret key and random symmetric encryption initialization vector pair. This pair is distinct for each chunk. A encrypts the random key with K_{SA}, as $e = Enc_{K_{SA}}(k_{\langle c \rangle})$. Next, A hashes the ciphertext C as $H(C)$. Subsequently, it computes its commitment to the encrypted chunk and the encrypted key as $T_{AS} = MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, \langle c \rangle, e, H(C), t]$ and sends the following to B.

$A \longrightarrow B$: [chunk response]$T_{AS}, \langle F \rangle, \langle c \rangle, e, C, t, p_A$

Step 7: To retrieve $k_{\langle c \rangle}$, B needs to request it from the server. As soon as B receives the encrypted chunk, B computes its own hash over the received ciphertext C' and forwards the following to S.

$B \longrightarrow S$:[decryption key request] $\langle A \rangle, \langle F \rangle, \langle c \rangle, e, H(C'), t, T_{AS}, p_A$

Step 8: If timestamp t is fresh enough, and p_A is not too much off, S checks if $T_{AS} = MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, \langle c \rangle, e, H(C'), t]$. The timestamp t freshness requirement forces B to expedite paying for decrypting the encrypted chunks. This fact allows A to promply acquire credit for its service. The ticket T_{AS} verification may fail either because $C' \neq C$ due to transmission error in step (6) or because A or B are misbehaving. Since S is unable to determine which is the case, it punishes neither AS or B and does not update their credit. It does not send the decryption key to B but it still notifies B of the discrepancy. In this case, B is

expected to disconnect from A and blacklist it in case A repeatedly sends invalid chunk response messages. If B keeps sending invalid decryption key requests, S penalizes him. If the verification succeeds, S checks whether B has sufficient credit to purchase the chunk c. It also checks again whether B has access to the content $\langle F \rangle$. If B is approved, it charges B and rewards A with Δ_c credit units. Subsequently, S decrypts $k'_{\langle c \rangle} = Dec_{K_{SA}}(e)$, and sends it to B.

$S \longrightarrow B$: [decryption key response] $\langle A \rangle, \langle F \rangle, \langle c \rangle, k'_{\langle c \rangle}$

B uses $k'_{\langle c \rangle}$ to decrypt the chunk as $c' = Dec_{k'_{\langle c \rangle}}(C')$. Next, we explain the complaint mechanism.

Step 9: If the decryption fails or if $H(c') \neq H(c)$ (step 2), B complains to S by sending the following message.

$B \longrightarrow S$:[complaint] $\langle A \rangle, \langle F \rangle, \langle c \rangle, T_{AS}, e, H(C'), t, p_A$

S ignores this message if *current-time* $> ts + T'$, where $T' > T$. $T' - T$ should be greater than the time needed for B to receive a decryption key response, decrypt the chunk and send a complaint to the server. With this condition, a misbehaving client A cannot avoid having complaints ruled against it, even if A ensures that the time elapsed between the moment A commits to the encrypted chunk and the moment the encrypted chunk is received by B is slightly less than T.

S also ignores the complaint message if a complaint for the same A and c is in a cache of recent complaints that S maintains for each client B. Complaints are evicted from this cache once *current-time* $> ts + T'$. If $T_{AS} \neq MAC_{K_{SA}}[\langle A \rangle, \langle B \rangle, \langle F \rangle, \langle c \rangle, e, H(C'), t]$, S punishes B. This is because S has already notified B in step (7) that T_{AS} is invalid. If T_{AS} verifies, S caches this complaint, recomputes K_{SA} as before, decrypts $k'_{\langle c \rangle} = Dec_{K_{SA}}(e)$ once again, retrieves c from its storage, and encrypts c himself using $k'_{\langle c \rangle}$, $C'' = Enc_{k'_{\langle c \rangle}}(c)$. If the hash of the ciphertext $H(C'')$ is equal to the value $H(C')$ that B sent to S, S decides that A has acted correctly and B's complaint is unjustified. Subsequently, S drops the complaint request and blacklists B. It also notifies A, which disconnects from B and blacklists it. Otherwise, if $H(C'') \neq H(C')$, S decides that B was cheated by A, removes A from its set of active clients, blacklists A, and revokes the corresponding credit charge on B. Similarly, B disconnects from A and blacklists it.

The server disconnects from a blacklisted client E, marks it as blacklisted in the credit file and denies access to E if it attempts to login. Future complaints concerning a blacklisted client E and for which T_{ES} verifies, are ruled against E without further processing.

Since a verdict on a complaint can adversely affect a client, each client needs to ensure that the commitments it generates are correct even in the rare case of a disk read error. Therefore, a client always verifies the read chunk against its hash before it encrypts the chunk and generates its commitment.

3.5 Other Protocol Issues

A content provider may be more concerned with scalability than it is with the free-riding problem presented in Section 2. In such case, it can deploy clients that use tit-for-tat incentives if their peers have content that interests them, i.e. the clients would upload plaintext content to peers that reciprocate with plaintext content. These clients would fall back to Dandelion's cryptographic fair-exchange mechanism when their peers do not have content that interests them. For example, selfish seeders would always upload encrypted content to their peers.

In case a client is unable to timely retrieve a missing chunk from its peers, it resorts to requesting the chunk from the server. If the server is not busy, it replies with the plaintext chunk. If it is moderately busy, it charges an appropriately large amount of credit $\Delta_s > \Delta_c$, sends the chunk and indicates that it is preferable for the client not to download chunks from the server. If the server is overloaded, it replies with an error message. Clients always download the content from the server in chunks, so that the system can seamlessly switch to the peer-serving mode when the server becomes busy.

Typically, a content distributor would deploy, in addition to the server, at least one client that possesses the complete content (initial seeder). In this way, the distributor ensures that the complete content is made available, even if the server is too busy to serve chunks.

4 Security Analysis

This section briefly lists the security properties of our design. For simplicity of presentation, we omit proofs on why these properties hold. They can be found in the Appendix of [51].

Lemma 4.1 If the server S charges a client B Δ_c credit units for a chunk c received from a selfish client A, B must have received the correct c, regardless of the actions taken by A.

Lemma 4.2 If a selfish client A always encrypts chunk c anew when servicing a request and if B gets correct c from A, then A is awarded Δ_c credit units from S, and B is charged Δ_c credit units from S.

Lemma 4.3 A selfish or a malicious client cannot assume another authorized client's A identity and issue messages under A, aiming at obtaining service at the

expense of A, charging A for service it did not obtain or causing A to be blacklisted. In addition, it cannot issue a valid T_{SA} for an invalid chunk that it sends to a client B and cause B to produce a complaint message that would result in a verdict against A.

Lemma 4.4 A malicious client cannot replay previously sent valid requests to the server or generate decryption key requests or complaints under A's ID, aiming at A being charged for service it did not obtain or being blacklisted because of invalid or duplicate complaints.

Observation 4.5 A client cannot download chunks from a selfish peer if it does not have sufficient credit. Our design choice to involve the server in every transaction, instead of using the fair exchange technique proposed in [43], enables the server to check a client's credit balance before the client retrieves the decryption key of a chunk.

Observation 4.6 To maintain an efficient content distribution pipeline, a client needs to relay a received chunk to its peers as soon as it receives it. However, the chunk may be invalid due to communication error or due to peer misbehavior. The performance of the system would be severely degraded if peers wasted bandwidth to relay invalid content. To address this issue, Dandelion clients send a decryption key request to the server immediately upon receiving the encrypted chunk. This design choice enables clients to promptly retrieve the chunk in its non-encrypted form. Thus, they can verify the chunk's integrity prior to uploading the chunk to their peers.

Observation 4.7 A malicious client cannot DoS attack the server by sending invalid content to other clients or repeatedly sending invalid complaints aiming at causing the server to perform the relatively expensive complaint validation. This is because it becomes blacklisted by both the server and its peers the moment the invalid complaint is ruled against it. In addition, a malicious client cannot attack the server by sending valid signed messages with redundant valid complaints. Our protocol detects duplicate complaints through the use of timestamps and caching of recent complaints.

Observation 4.8 A malicious client B can always abandon any instance of the protocol. In such case, A does not receive any credit, as argued in Lemmas 4.1 to 4.3, even though B may have consumed A's resources. This is a denial of service attack against A. Note that this attack would require that the malicious client B expends resources proportional to the resources of the victim A, therefore it is not particularly practical. Nevertheless, we prevent blacklisted clients or clients that do not maintain paid accounts with the content provider from launching such attack by having S issue a short-lived ticket T_{SA} to authorized clients only. T_{SA} is checked for validity by A (steps 4 and 6 in Section 3.4.2). In addition, S may charge an authorized B for the issuance of tickets T_{SA} effectively deterring B from maliciously expending both A's and the server's resources.

Owing to properties 4.1, 4.2, 4.3 and 4.5, and given that the content provider employs appropriate pricing schemes, Dandelion ensures that selfish (rational) clients increase their utility when they upload correct chunks and obtain virtual currency, while misbehaving clients cannot increase their utility. Consequently, Dandelion entices selfish clients to upload to their peers, resulting in a Nash equilibrium of cooperation.

5 Implementation

This section describes a prototype C implementation of Dandelion that is suitable for cooperative content distribution of static content. It uses the *openssl* [5] library for cryptographic operations and standard file I/O system calls to efficiently manage credit information, which is stored in a simple file.

5.1 Server Implementation

The server and the credit base are logical modules and could be distributed over a cluster to improve scalability. For simplicity, our current implementation combines the content provider and the credit base at a single server.

The server implementation is single-threaded and event-driven. The network operations are asynchronous, and data are transmitted over TCP.

Each client is assigned an entry in a credit file, which stores the credit as well as authentication and file access control information. Each entry has the same size and the client ID determines the offset of the entry of each client in the file, thus each entry can be efficiently accessed for both queries and updates.

The server queries and updates a client's credit from and to the credit file upon every transaction, Yet, it does not force commitment of the update to persistent storage. Instead, it relies on the OS to perform the commitment. If the server application crashes, the update will still be copied from the kernel buffer to persistent storage. Still, the OS may crash or the server may lose power before the updated data have been committed. However, in practice, a typical Dandelion deployment would run a stable operating system and use backup power supply. In addition, the server could mirror the credit base on multiple machines using high speed IP/Ethernet I/O. In addition, transactions would not involve very large amounts of money per user. Hence, we believe it is preferable not to incur the high cost of committing the credit updates

to non-volatile memory after every transaction or after a few transactions (operations 12 and 13 in Table 1).

5.2 Client Implementation

The client side is also single-threaded and event-driven. A client may leech or seed multiple files at a time. A client can be decomposed into two logical modules: a) the *connection management* module; and b) the *peer-serving* module.

The connection management module performs *peering* and *uploader discovery*. With peering, each client obtains a random partial swarm view from the server and strives to connect to $O(\log n)$ peers, where n is the number of nodes in the Dandelion swarm, as communicated to the node by the server. As a result, the swarm approximates a random graph with logarithmic out-degree, which has been shown to have high connectivity[21]. With uploader discovery, a client attempts to remain connected to a minimum number of uploading peers. If the number of recent uploaders drops below a threshold, a client requests from the server a new swarm view and connects to the peers in the new view.

The peer-serving module performs *content reconciliation* and *downloader selection*. Content reconciliation refers to the client functionality for announcing recently received chunks, requesting missing chunks, requesting decryption keys for received encrypted chunks, and replying to chunk requests. Our implementation employs locally-rarest-random [39] scheduling in requesting missing chunks from clients. To efficiently utilize their downlink bandwidth using TCP, clients strive to at all times keep a specified number of outstanding chunk requests [26, 40], which have been sent to a peer and have not been responded to.

With downloader selection, the system aims at making chunks available to the network as soon as possible. In the following description, n denotes the number of parallel downloaders. Dandelion's downloader selection algorithm is similar to the *seeder* choking algorithm used in the "vanilla" BitTorrent 4.0.2, as documented in [41]. The algorithm is executed by each client every 10 seconds. It is also executed when a when a peer that is selected to be downloader disconnects. The algorithm proceeds as follows: a) peers that are interested in the client's content are ranked based on the time they were last selected to be downloaders (most recent first); b) the client selects as downloaders the $n-2$ top ranked peers; c) in case of a tie, the peer with the highest download rate from the client is ranked higher; and d) the client randomly selects an additional downloader from the non-selected nodes that are interested in the client's content. Step (d) is performed in expectation of discovering a fast downloader and to jumpstart peers that recently joined the swarm.

This downloader selection algorithm aims at reducing the amount of duplicate data a client needs to upload, before it has uploaded a full copy of its content to the swarm. Downloader selection improves the system's performance in the following additional ways. First, it limits the number of peers a client concurrently uploads to, such that complete chunks are made available to other peers and relayed by them at faster rates. Second, given that all clients are connected to roughly the same number of peers, it also limits the number of peers a client concurrently downloads from to approximately n. As a result, the rate with which the client downloads complete chunks increases. Last, by limiting the number of connections over which clients upload, it avoids the inefficiency and unfairness that is observed when many TCP flows share a bottleneck link [46].

The number of peers that download from a client in parallel depends on the client's upload bandwidth. We have empirically determined that a good value for this parameter in the PlanetLab environment is 10.

6 Evaluation

The goal of our experimental evaluation is to demonstrate the viability and to identify the scalability limits of Dandelion's centralized and non-manipulable virtual-currency-based incentives.

6.1 Dandelion Profiling

We profile the cost of operations performed by the server aiming at identifying the performance bottlenecks of our design. The measurements are performed on a dual Pentium D 2.8GHZ/1MB CPU with 1GB RAM and 250GB/7200RPM HDD running Linux 2.6.5-1.358smp.

Table 1 lists the cost of per chunk Dandelion operations. In a flash crowd event, the main task of a Dandelion server is to: a) receive the decryption key request (operation 7); b) authenticate the request by computing an HMAC (operation 1); c) verify the ticket by computing an HMAC (operation 2); d) decrypt the encrypted decryption key (operation 3); e) query and update the credit of the two clients involved (operations 10 and 11); f) sign the decryption key response (operation 4); and g) send the decryption key response (operation 8).

As can be seen in the table, the per chunk cryptographic operations of the server (operations 1-4) are highly efficient (total 109 usec), as only symmetric cryptography is employed. The credit management operations (10 and 11) are also efficient (total 24 usec). On the other hand, the communication costs clearly dominate the processing costs, indicating that for 1Mb/s uplink and downlink, the downlink is the bottleneck.

The cost of a complaint is higher because in addition to verifying a ticket, it involves reading a chunk, encrypting

	Dandelion operation	Size	Time (ms)
	CPU-centric Operation		
1	Authenticate decryption key request	98 bytes	.018
2	Generate ticket for decryption key key request or complaint verification	78 bytes	.018
3	Encrypt/decrypt decryption key	32 bytes	.056
4	Sign decryption key response	46 bytes	.017
5	Encrypt/decrypt chunk	128 KB	.715
6	Hash encrypted chunk for commitment generation or for commitment processing	128 KB	.487
	Communication Operation		
7	Receive decryption key request	156 bytes	~1.79
8	Transmit decryption key response	104 bytes	~1.39
9	Transmit chunk	128 KB	~1053
	Credit Management Operation		
10	Query credit file	N/A	~0.004
11	Update credit file without commit to disk (rely on OS)	N/A	~0.02
12	Update credit file and commit to disk	N/A	~9.25
13	Update credit file and commit to disk every 100 updates	N/A	~0.27

Table 1: **Timings of per chunk transaction Dandelion operations. Timings for operations 1-6 are obtained using *getrusage(RUSAGE_SELF)* over 10000 executions to obtain 1 usec precision. Timings for operations 7-9 are approximated according to our application layer rate-limiting for 1Mb/s uplink and 1Mb/s downlink. They are provided as reference for comparison with CPU-centric and credit management operations. Timings for operations 10-13 are approximated using *gettimeofday()* over 10000 executions. Operations 3 and 5 use 8-byte-block Blowfish-CBC with 128-bit key and 128-bit initialization vector.1, 2 and 4 use HMAC-SHA1 with 128-bit key. Operation 6 uses SHA-1. Operations 10-12 are performed on a credit file with 10000 44-byte entries. For committing to disk in operations 12 and 13, we use fsync() and we disable HDD caching.**

it with the sender client's key (operation 5), and hashing the encrypted chunk (operation 6).

Note that the profiling of the server repeats the same operation multiple times. It does not consider the parallel processing of I/O and CPU operations. In addition, it does not include the cost of system calls and the cost of TCP/IP stack processing. Therefore, we refrain from deriving conclusions on the throughput of the server. Such conclusions are derived in the subsequent evaluation.

6.2 Server Performance

A Dandelion server mediates the chunk exchanges between its clients. The download throughput of clients in our system is bound by how fast a server can process their decryption key requests. Both the server's computational resources and bandwidth may become the performance bottleneck.

We deploy a Dandelion server that runs on the same machine as the one used for Dandelion profiling. We also deploy ~200 clients that run on distinct PlanetLab hosts. The server machine shares a 100Mb/s Ethernet II link. To mitigate bandwidth variability in the shared link and to emulate a low cost server with an uplink and downlink that ranges from 1Mb/s to 5Mb/s, we rate-limit the server at the application layer.

In each experiment, the clients send requests for decryption keys to the server and we measure the aggregate rate with which all clients receive key responses. The server always queries and updates the credit base from and to the credit file without forcing commitment to disk. The specified per client request rate varies from 1 to 14 requests per second. For each request rate, the experiment duration was 10 minutes and the results were averaged over 10 runs. As the request rate increases and the server's receiver buffers become full, clients do not send new requests at the specified rate, due to TCP's flow control. When the request rate increases to the point that the server's resources become saturated, the key response rate from the server decreases.

Figure 3(a) depicts the server's decryption key throughput for various server bandwidth capacities. As the bandwidth increases from 1Mb/s to 3Mb/s, the server's decryption key response throughput increases. This indicates that for 1Mb/s to 3Mb/s access links, the bottleneck is the bandwidth. When the bandwidth limit is 4Mb/s and 5Mb/s, the server exhibits similar performance, which suggests that the access link is not saturated at 4Mb/s. The results show that a server running on our commodity PC with 4Mb/s or 5Mb/s access link can process up to ~1200 decryption key requests per second. This indicates that with a 128KB chunk size, this server may simultaneously support almost 1200 clients that download from each other at 128KB/s. With a larger chunk size, each such client sends decryption key requests at a slower rate, and the number of supported clients increases.

Figures 3(b) and 3(c) show the average CPU and memory utilization at the server over the duration of the above experiments. We observe that for 4Mb/s and 5Mb/s, the server's CPU utilization reaches ~100%, indicating that the bottleneck is the CPU. In Figure 3(c), we see that the server consumes a very small portion of the available memory.

6.3 System Performance

The following experiments evaluate the performance of the Dandelion system on PlanetLab. We examine the impact of chunk size and the impact of seeding on the performance of the system. We also compare our system's performance to BitTorrent's. In all experiments we ran a Dandelion server within a PlanetLab VServer

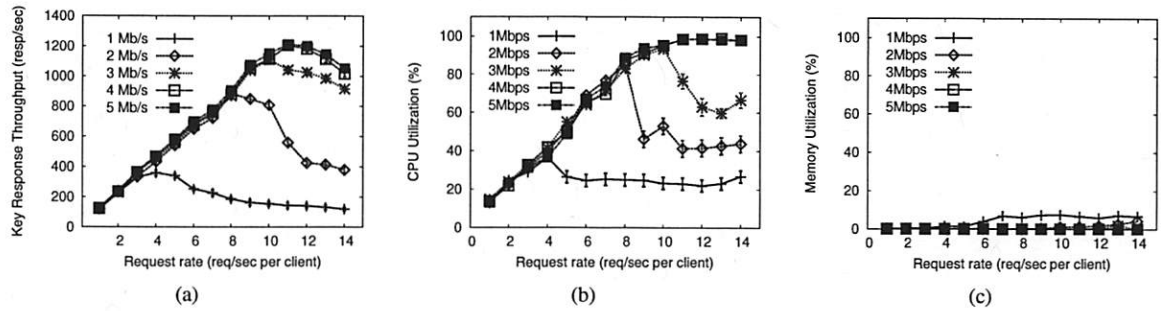


Figure 3: **(a) Aggregate decryption key response throughput of the Dandelion server as a function of per-client key request rate, for varying server rate-limits; (b) server's CPU utilization(%) as a function of per-client decryption key request rate; (c) server's memory utilization (%) as a function of per-client decryption key request rate.**

spawned on a highly available Xeon 3GHZ/2MB CPU and 2GB RAM machine. We rate-limit the server at 2Mb/s.

Leechers are given sufficient initial credit to completely download a file. Clients always respond to chunk requests from their selected downloaders and they never request chunks from the server. We do not rate-limit the Dandelion and BitTorrent clients, as a means for testing our system in heterogeneous Internet environments. To cover the bandwidth-delay product in PlanetLab, the TCP sender and receiver buffer size is set equal to 120KB.

For each configuration we repeat the experiment 10 times and we extract mean values and 95% confidence intervals over the swarm-wide mean file download completion times of each run. The file download completion time is the time that elapses between the moment the client contacts the server to start a download and the moment its download is completed.

6.3.1 Selecting Chunk Size

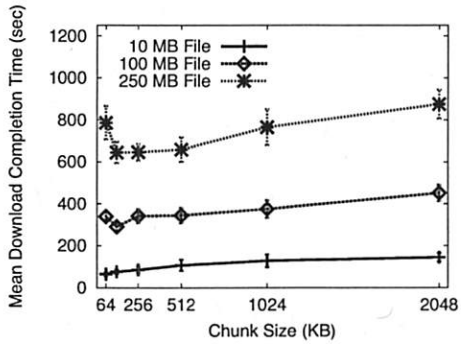


Figure 4: **Mean file download completion times of 40 leechers as a function of chunk size. The swarm has one initial seeder. The x axis corresponds to the chunk size. The y axis corresponds to the mean download completion time in the swarm. The error bars correspond to 95% confidence intervals.**

This experiment aims at examining the tradeoffs involved in selecting the size of the chunk, the verifiable transaction unit in Dandelion. Intuitively, since clients are able to serve a chunk only as soon as they complete its download, a smaller chunk size yields a more efficient distribution pipeline. In addition, when the file is divided into many pieces, chunk scheduling techniques such as rarest-first can be more effective, as there is sufficient content entropy in the network. Consequently, clients can promptly discover and download content of interest. However, a smaller chunk size increases the rate with which key requests are sent to the server, reducing the scalability of the system. In addition, due to TCP's slow start, a small chunk size cannot ensure high bandwidth utilization during the TCP transfer of any chunk.

In each configuration, we deploy around 40 Dandelion leechers and one initial seeder, i.e. a client that has the complete file before the distribution starts. Leechers start downloading files almost simultaneously. We deploy only 40 leechers to ensure that the server is not saturated, even if we use 64KB chunk size.

Figure 4 shows the leecher mean download completion time as a function of the chunk size. For smaller files, e.g., the 10MB file, the system has the best performance for chunk size equal to 64KB. The system's performance degrades with the chunk size. As the file size increases, the beneficial impact of small chunks, becomes less significant. For example, for 250MB file, a 128KB chunk size yields notably better performance than a 64KB chunk size.

Based on the above results and further fine-tuning, in the rest of this evaluation, we use 128KB chunks.

6.3.2 Impact of Dandelion Seeders

One of Dandelion's main advantages is that it provides robust incentives for clients to seed. We quantify the performance gains from the existence of seeders in our system. In each experiment, we deploy ~200 leech-

ers. Leechers start downloading the file almost simultaneously, creating a flash crowd.

We show the impact of seeders by varying the probability that a leecher remains online to seed a file after it completes its download. In each experiment, a swarm has one initial seeder. Upon completion of its download, each leecher stays in the swarm and seeds with probability a. Probability a varies in 25% and 100%.

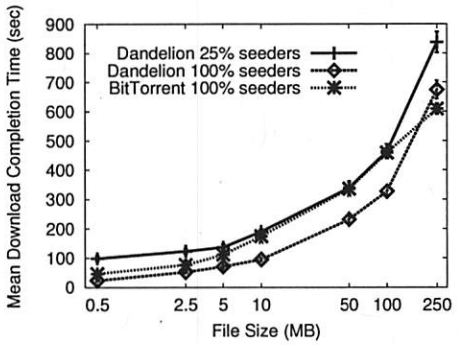


Figure 5: **Swarm-wide mean file download completion times of ~200 leechers as a function of file size for varying portion of leechers that become seeders. The error bars correspond to 95% confidence intervals.**

Figure 5 depicts the mean download completion time over all ~200 leechers as a function of the file size, for varying a. The results show the beneficial impact of seeders and the importance of a mechanism to robustly incent seeding. For example, for a 250MB file, we observe a swarm-wide mean download completion time of 674 sec and 837 sec when leechers become seeders with 100% and 25% probability, respectively. If we express the impact of seeders as the ratio of the mean download time for $a = 100\%$ over the mean download time for $a = 25\%$, we observe that the impact is reduced as the file size increases. The larger the file is, the longer clients remain online to download it, resulting in clients contributing their upload bandwidth for longer periods. For smaller files however, leechers rely heavily on the initial seeder and the leechers that become seeders to download their content from. Therefore for small files, a reduction in probability a results in substantially longer download completion times.

6.3.3 Comparison with BitTorrent

Figure 5 also shows the download completion times of ~200 tit-for-tat compliant CTorrent 1.3.4 leechers. All BitTorrent leechers remain online after their download completion ($a = 100\%$). The purpose of this illustration is to show that Dandelion can attain performance comparable to the one achieved by BitTorrent, although it employs a different downloader selection algorithm and involves the server in each chunk exchange.

Although Dandelion appears to outperform BitTorrent for certain file sizes, we do not claim that it is in general a better-performing protocol. The performance of both protocols is highly dependent on numerous parameters, which we have not exhaustively analyzed.

6.3.4 Credit Distribution

We examine the distribution of credit during a Dandelion file distribution. The purpose of this measurement is to identify which type of clients tend to accumulate the most credit in swarms of similar configuration to ours.

Figure 6 shows the scatter plot of the client's credit after a single 250MB file download by ~200 leechers together with the mean download rate of each client. In the experiment, there is only one initial seeder. All nodes are given 100% of the credit needed to download the file and they all become seeders upon download completion. We observe that the seeder obtained the most credit during the file distribution. This is expected, as a seeder is always in position to upload useful content to its peers and our seeder had a fast access link. Since fast downloaders obtain useful content earlier in the distribution and are likely to have uplinks proportional to their downlink, they should be able to deliver more content and earn more credit. Our results confirm this intuition and show that there is *strong* correlation between average download rate and credit ratio, i.e. the product-moment correlation coefficient is equal to 0.72.

We also observe that many clients uploaded substantially less than they downloaded. Indicatively, 25.8% of the clients had less than 70% of their initial credit.

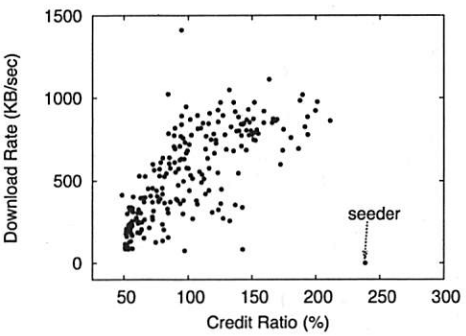


Figure 6: **The scatter plot of the distribution of credit after ~200 leechers have completed a 250MB file download and their average download rates. The x axis shows the credit ratio, which is the ratio of the remaining credit of a client over its initial credit. The y axis shows the average download rate of each client. The seeder is included for illustration purposes, but its download rate is invalid.**

7 Conclusion

This paper describes Dandelion: a cooperative (P2P) system for the distribution of paid content. Dandelion's

primary function is to enable a content provider to provide strong incentives for clients to contribute their uplink bandwidth.

Dandelion rewards selfish clients with virtual currency (credit) when they upload valid content to their peers and charges clients when they download content from selfish peers. Since Dandelion employs a non-manipulable cryptographic scheme for the fair exchange of content uploads for credit, the content provider is able to redeem a client's credit for monetary rewards. Thus, it provides strong incentives for clients to seed content and to not free-ride.

Our experimental results show that seeding substantially improves swarm-wide performance. They also show that a Dandelion server running on commodity hardware and with moderate bandwidth can scale to a few thousand clients. Dandelion's deployment in medium size swarms demonstrates that it can attain performance that is comparable to BitTorrent. These facts demonstrate the plausibility of our design choice: centralizing the incentive mechanism in order to increase resource availability in P2P content distribution networks.

8 Acknowledgements

We are thankful to Nikitas Liogkas, Eddie Kohler and the anonymous reviewers for their extensive and fruitful feedback. We also thank Rex Chen, Dehn Sy and Lichun Bao with Calit2 for providing space and equipment for our experiments. This work was supported in part by NSF award CNS-0627166.

References

[1] http://www.ietf.org/rfc/rfc4346.txt.
[2] Enhanced CTorrent. www.rahul.net/dholmes/ctorrent/.
[3] Kazaa. www.kazaa.com.
[4] Kazaa Lite. en.wikipedia.org/wiki/KazaaLite.
[5] Openssl. www.openssl.org.
[6] Peer exchange. www.azureuswiki.com/index.php/Peer_Exchange.
[7] The eMule Project. www.emule-project.net.
[8] The MNet Project. mnetproject.org.
[9] Trackerless BitTorrent. www.bittorrent.com/trackerless.html.
[10] iTunes outsells CD stores as digital revolution gathers pace. arts.guardian.co.uk/netmusic/story/0,,1649421,00.html, 2005.
[11] BitTorrent Strikes Digital Download Deals with 20th Century Fox, G4, Kadokawa, Lionsgate, MTV Networks, Palm Pictures, Paramount and Starz Media. home.businesswire.com/portal/site/google/index.jsp?ndmViewId=news_view&newsId=20061128006262&newsLang=en, 2006.
[12] EMI Music makes its catalog available to Qtrax: the world's first ad-supported, legitimate P2P service. www.emigroup.com/Press/2006/press25.htm, June 2006.
[13] Music denied - Shoppers overwhelm iTunes. edition.cnn.com/2006/TECH/internet/12/28/itunes.slowdown.ap/index.html?eref=rss_topstories, 2006.
[14] Quote from PACIFIC BELL: $18000 per month for an OC3 line. shopforoc3.com/, Mar. 2006.
[15] Universal announces a new Download-to-own service. edition.cnn.com/2006/TECH/03/23/movie.download/index.html, Mar 2006.
[16] Warner Bros to sell films via BitTorrent. www.msnbc.msn.com/id/12694081/, June 2006.
[17] N. Andrade, M. Mowbray, A. Lima, G. Wagner, and M. Ripeanu. Influences on Cooperation in Bittorrent Communities. In *P2P Econ*, 2005.
[18] N. Asokan, M. Schunter, and M. Waidner. Optimistic Protocols for Fair Exchange. In *CCS*, 1997.
[19] F. Bao, R. Deng, and W. Mao. Efficient and Practical Fair Exchange Protocols with Off-line TTP. In *S&P*, 1998.
[20] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. In *LNCS*, volume 1109, 1996.
[21] B. Bollobas. Random Graphs. In *Academic Press*, 1985.
[22] E. F. Brickell, D. Chaum, I. Damg, and J. V. de Graaf. Gradual and Verifiable Release of a Secret. In *CRYPTO*, 1988.
[23] B. Cheng, X. Liu, Z. Zhang, and H. Jin. A measurement study of a peer-to-peer video-on-demand system. In *IPTPS*, 2007.
[24] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an Overlay Testbed for Broad-coverage Services. In *SIGCOMM CCR*, 2003.
[25] R. Cleve. Controlled Gradual Disclosure schemes for Random bits and their Applications. In *CRYPTO*, 1989.
[26] B. Cohen. Incentives Build Robustness in BitTorrent. In *P2P Econ*, 2003.
[27] I. B. Damg. Practical and Provably Secure Release of a Secret and Exchange of Signatures. In *EUROCRYPT*, 1994.
[28] J. R. Douceur. The Sybil Attack. In *IPTPS*, 2002.
[29] P. Druschel, A. Nandi, T.-W. J. Ngan, A. Singh, and D. Wallach. Scrivener: Providing Incentives in Cooperative Content Distribution Systems. In *Middleware*, 2005.
[30] S. Even, O. Goldreich, and A. Lempel. A Randomized Protocol for Signing Contracts. In *Communications of the ACM*, 1985.
[31] B. Fan, D.-M. Chiu, and J. C. Lui. The Delicate Tradeoffs in BitTorrent-like File Sharing Protocol Design. In *ICNP*, 2006.
[32] K. Franklin and M. K. Reiter. Fair exchange with a semi-trusted third party. In *CCS*, 1997.
[33] M. K. Franklin and G. Tsudik. Secure Group Barter: Multi-party Fair Exchange with Semi-Trusted Neutral Parties. In *Financial Cryptography*, 1998.
[34] J. Gray. Distributed Computing Economics. Technical report, Microsoft Research, 2003. MSR-TR-2003-24.
[35] B. Horne, B. Pinkas, and T. Sander. Escrow Services and Incentives in Peer-to-peer networks. In *EC*, 2001.
[36] D. Hughes, G. Coulson, and J. Walkerdine. Free Riding on Gnutella Revisited: The Bell Tolls? In *IEEE Distributed Systems Online*, 2005.
[37] S. Jun and M. Ahamad. Incentives in BitTorrent Induce Free Riding. In *P2P Econ*, 2005.
[38] I. Keidar, R. Melamed, and A. Orda. EquiCast: Scalable Multicast with Selfish Users. In *PODS*, 2006.
[39] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In *SOSP*, 2003.
[40] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Maintaining High Bandwidth Under Dynamic Network Conditions. In *USENIX*, 2005.
[41] A. Legout, N. Liogkas, E. Kohler, and L. Zhang. Clustering and Sharing Incentives in BitTorrent Systems. SIGMETRICS, 2007.
[42] H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR Gossip. In *OSDI*, 2006.
[43] J. Li and X. Kang. Proof of Service in a Hybrid P2P Environment. In *ISPA Workshops*, 2005.
[44] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang. Exploiting BitTorrent For Fun (But Not Profit). In *IPTPS*, 2006.
[45] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free Riding in BitTorrent is Cheap. In *HotNets*, November 2006.
[46] R. Morris. TCP Behavior with Many Flows. In *ICNP*, 1997.
[47] T. Okamoto and K. Ohta. How to Simultaneously Exchange Secrets by General Assumptions. In *CCS*, 1994.
[48] B. Schneier. Applied Cryptography, 2nd edition, 1995.
[49] J. Shneidman, D. Parkes, and L. Massoulie. Faithfulness in Internet Algorithms. In *PINS*, 2004.
[50] M. Sirivianos, J. H. Park, R. Chen, and X. Yang. Free-riding in BitTorrent Networks with the Large View Exploit. In *IPTPS*, www.ics.uci.edu/~msirivia/publications/large-view.pdf, 2007.
[51] M. Sirivianos, X. Yang, and S. Jarecki. Dandelion: Cooperative Content Distribution with Robust Incentives. In *NetEcon*, www.ics.uci.edu/~msirivia/publications/dandelion-netecon.pdf, 2006.
[52] K. Tamilmani, V. Pai, and A. Mohr. SWIFT: A System with Incentives for Trading. In *P2P Econ*, 2004.
[53] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A Secure Economic Framework for P2P Resource Sharing. In *P2P Econ*, 2003.
[54] K. Wei, Y.-F. Chen, A. J. Smith, and B. Vo. WhoPay: a Scalable and Anonymous Payment System for Peer-to-Peer Environments. In *ICDCS*, 2006.
[55] B. Yang and H. Garcia-Molina. PPay: Micropayments for Peer-to-peer Systems. In *CCS*, 2003.
[56] J. Zhou and D. Gollmann. A Fair Non-repudiation Protocol. In *S&P*, 1996.
[57] J. Zhou and D. Gollmann. An Efficient Non-repudiation Protocol. In *CSFW*, 1996.

Using Provenance to Aid in Personal File Search

Sam Shah[*] Craig A. N. Soules[†] Gregory R. Ganger[‡] Brian D. Noble[*]

[*]*University of Michigan* [†]*HP Labs* [‡]*Carnegie Mellon University*

Abstract

As the scope of personal data grows, it becomes increasingly difficult to find what we need when we need it. Desktop search tools provide a potential answer, but most existing tools are incomplete solutions: they index content, but fail to capture dynamic relationships from the user's context. One emerging solution to this is context-enhanced search, a technique that reorders and extends the results of content-only search using contextual information. Within this framework, we propose using strict *causality*, rather than temporal locality, the current state of the art, to direct contextual searches. Causality more accurately identifies data flow between files, reducing the false-positives created by context-switching and background noise. Further, unlike previous work, we conduct an online user study with a fully-functioning implementation to evaluate *user-perceived* search quality directly. Search results generated by our causality mechanism are rated a statistically-significant 17% higher on average over all queries than by using content-only search or context-enhanced search with temporal locality.

1 Introduction

Personal data has become increasingly hard to manage, find, and retrieve as its scope has grown. As storage capacity continues to increase, the number of files belonging to an individual user, whether a home or corporate desktop user, has increased accordingly [7]. The principle challenge is no longer efficiently storing this data, but rather organizing it. To reduce the friction users experience in finding their data, many personal search tools have emerged. These tools build a content index and allow keyword search across this index.

Despite their growing prevalence, most of these tools are, however, incomplete solutions: they index content, not context. They capture only static, syntactic relationships, not dynamic, semantic ones. To see why this is important, consider the difference between compiler optimization and branch prediction. The compiler has access only to the code, while the processor can see how that code is commonly used. Just as run-time information leads to significant performance optimizations, users find contextual and semantic information useful in searching their own repositories [22].

Context-enhanced search is beginning to receive attention, but it is unclear what dynamic information is most useful in assisting search. Soules and Ganger [21] developed a system, named Connections, that uses *temporal locality* to capture the provenance of data: for each new file written, the set of files read "recently" form a kinship or *relation graph*, which Connections uses to extend search results generated by traditional static, content-based indexing tools. Temporal locality is likely to capture many true relationships, but may also capture spurious, coincidental ones. For example, a user who listens to music while authoring a document in her word processor may or may not consider the two "related" when searching for a specific document.

To capture the benefit of temporal locality while avoiding its pitfall, we provide a different mechanism to deduce provenance: *causality*. That is, we use data flow through and between applications to impart a more accurate relation graph. We show that this yields more desirable search results than either content-only indexing or kinship induced by temporal locality.

Our context-enhancing search has been implemented for Windows platforms. As part of our evaluation, we conduct a user study with this prototype implementation to measure a user's perceived search quality directly. To accomplish this, we adapt two common techniques from the social sciences and human-computer interaction to the area of personal file search. First, we conduct a randomized, controlled trial to gauge the end-to-end effects of our indexing technique. Second, we conduct a repeated measures experiment, where users evaluate the different indexing techniques side-by-side, locally on their own machines. This style of experiment is methodologically superior as it measures quality

directly while preserving privacy of user data and actions.

The results indicate that our causal provenance algorithm fares better than using temporal locality or pure content-only search, being rated a statistically-significant 17% higher, on average, than the other algorithms by users with minimal space and time overheads. Further, as part of our study, we also provide some statistics about personal search behavior.

The contributions of this paper are:

1. The identification of *causality* as a useful mechanism to inform contextual indexing tools and a description of a prototype system for capturing it.
2. An exploration of the search behavior of a population of 27 users over a period of one month.
3. A user study, including a methodology for evaluating personal search systems, demonstrating that causality-enhanced indexing provides higher quality search results than either those based on temporal locality or those using content information only.

The remainder of this paper is organized as follows. In Section 2, we give an overview of related work. Section 3 describes how our system deduces and uses kinship relationships, with Section 4 outlining our prototype implementation. Section 5 motivates and presents our evaluation and user study and Section 6 explores the search behavior of our sample population. Finally, Section 7 concludes.

2 Related Work

There are various static indexing tools for one's filespace. Instead of strict hierarchal naming, the semantic file system [10] allows assignment of attributes to files, facilitating search over these attributes. Since most users are averse to ascribing keywords to their files, the semantic file system provides transducers to distill file contents into keywords. The semantic file system focuses on the mechanism to store attributes, not on content analysis to distill these attributes.

There are several content-based search tools available today, including Google Desktop Search, Windows Desktop Search and Yahoo! Desktop Search, among others. These systems extract a file's content into an index, permitting search across this index. While the details of such systems are opaque, it is likely they use forefront technologies from the information retrieval community. Several such advanced research systems exist, Indri [1] being a prime example. These tools are orthogonal to our system in that they all analyze static data with well-defined types to generate an index, ignoring crucial contextual information that establishes semantic relationships between files.

The seminal work in using gathered context to aid in file search is by Soules and Ganger [21] in the form of a file system search tool named Connections. Connections identifies temporal relationships between files and uses that information to expand and reorder traditional content-only search results, improving average precision and recall compared to Indri. We use some component algorithms from Connections (§3.2) and compare against its temporal locality approach (§3.1.1).

Our notion of provenance is a subset of that used by the provenance-aware storage system (PASS) [17]. PASS attempts to capture a complete lineage of a file, including the system environment and user- and application-specified annotations of provenance. A PASS filesystem, if available, would negate the need for our relation graph. Indeed, the technique used by PASS to capture system-level provenance is very similar to our causality algorithm (§3.1.2).

Several systems leverage other forms of context for file organization and search. Phlat [5] is a user interface for personal search, running on Windows Desktop Search, that also provides a mechanism for tagging or classifying of data. The user can search and filter by contextual cues such as date and person. Our system provides a simpler UI, permitting search by keywords only (§4), but could use Phlat's interface in the future. Another system, called "Stuff I've Seen" [8], remembers previously seen information, providing an interface that allows a user to search their historical information using contextual cues. The Haystack project [12] is a personal information manager that organizes data, and operations on data, in a context-sensitive manner. Lifestreams [9] provides an interface that uses time as its indexing and presentation mechanism, essentially ordering results by last access time. Our provenance techniques could enhance these systems through automated clustering of semantically-related items.

3 Architecture

Our architecture matches that of Soules and Ganger [21]: we augment traditional content search using kinship relations between files. After the user enters keywords in our search tool, the tool runs traditional content-only search using those keywords—the *content-only phase*—and then uses the previously constructed relation graph to reorder these results and identify additional hits—the *context-enhancing phase*. These new results are then returned to the user. Background tasks run on the user's machine to periodically index a file's content for the content-only phase and to monitor system events to build the relation graph for the context-enhancing phase. This section describes how the system deduces and uses these relationships to re-rank results.

3.1 Inferring Kinship Relationships

A kinship relation, $f \rightarrow f'$ where f and f' are files on a user's system, indicates that f is an ancestor of f',

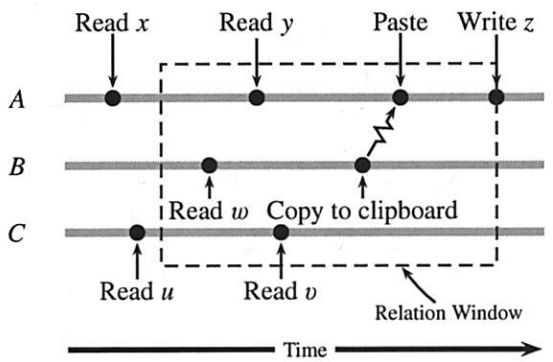


Figure 1. A time diagram of system events used to illustrate the differences between the provenance algorithms.

implying that f may have played a role in the origin of f'. These relationships are encoded in the relation graph, which is used to reorder and extend search results in the context-enhancing phase.

We evaluate two methods of deducing these kinship relations: temporal locality and causality. Both methods classify the source file of a read as input and the destination file of a write as output by inferring user task behavior from observed actions.

3.1.1 Temporal Locality Algorithm

The temporal locality algorithm, as employed in Soules and Ganger [21], infers relations by maintaining a sliding *relation window* of files accessed within the previous t seconds system-wide. Any write operation within this window is tied to any previous read operation within the window. This is known as the read/write operational filter with directed links in Soules and Ganger [21], which was found the most effective of several considered.

Consider the sequence of system events shown in Figure 1. There are three processes, A, B, and C, running concurrently. C reads files u and v, A reads files x and y. B reads w and copies data to A through a clipboard IPC action initiated by the user. Following this, A then writes file z.

The relation window at z's write contains reads of y, w, and v. The temporal locality algorithm is process agnostic and views reads and writes system-wide, distinguishing only between users. The algorithm thus returns the relations $\{y \rightarrow z, w \rightarrow z, v \rightarrow z\}$.

The relation window attempts to capture the transient nature of a user task. Too long a window will cause unrelated tasks to be grouped, but too short a window will cause relationships to be missed.

3.1.2 Causality Algorithm

Rather than using a sliding window to deduce user tasks, this paper proposes viewing each process as a filter that mutates its input to produce some output. This causality algorithm tracks how input flows—at the granularity of processes—to construct kinship relations, determining what output is causally related to which inputs.

Specifically, whenever a write event occurs, the following relations are formed:

(a) Any previous files read within the same process are tied to the current file being written;
(b) Further, the algorithm tracks IPC transmits and its corresponding receives, forming additional relationships by assessing the transitive closure of file system events formed across these IPC boundaries.

That is, for each relation $f \rightarrow f'$, there is a directed left-to-right path in the time diagram starting at a read event of file f and ending at the write of file f'. There is no temporal bound within this algorithm.

Reconsidering Figure 1, A reads x and y to generate z; the causality algorithm produces the relations $\{x \rightarrow z, y \rightarrow z\}$ via condition (a). B produces no output files given its read of w, but the copy-and-paste operation represents an IPC transmit from B with a corresponding receive in A. By condition (b), this causes the relation $w \rightarrow z$ to be made. C's reads are dismissed as they do not influence the write of z or any other data.

Causality forms fewer relationships than temporal locality, avoiding many false relationships. Unrelated tasks happening concurrently are more likely to be deemed related under temporal locality, while causality is more conservative. Further, when a user switches between disparate tasks, the temporary period where incorrect relations form under temporal locality is mitigated by the causality algorithm.

A user working on a spreadsheet with her music player in the background may form spurious relationships between her music files and her document under temporal locality, but not under causality; those tasks are distinct processes and no data is shared. Additionally, if she switches to her email client and saves an attachment, her spreadsheet may be an ancestor of that attachment under temporal locality if the file system events coincide within the relation window.

Long-lived processes are a mixed bag. A user opening a document in a word processor, writing for the afternoon, then saving it under a new name would lose the association with the original document under temporal locality, but not causality. A user working with her text editor to author several unrelated documents within the same process would have spurious relations formed with causality, but perhaps not with temporal locality.

Causality can fare worse under situations where data transfer occurs through hidden channels due to loss of real context. This is most evident when a user exercises her brain as the "clipboard," such as when she reads a value off a spreadsheet and then keys it manually into her

document. As future work, we are investigating using window focus to demarcate user tasks [18] as a means to group related processes together and capture these hidden channels.

3.1.3 Relation Graph

Relations formed are encoded in the relation graph: a directed graph whose vertices represent files on a user's system with edges constituting a kinship relation between files and the weight of that edge representing the strength of the bond. The edge's direction represents an input file to an output file.

For each relation of the form $f \rightarrow f'$, the relation graph consists of an edge from vertex f to vertex f' with the edge weight equalling the count of $f \rightarrow f'$ relations seen. To prevent heavy weightings due to consecutive writes to a single file, successive write events are coalesced into a single event in both algorithms.

3.2 Reranking and Extending Results

After a query is issued, the tool first runs traditional content-only search using keywords given by the user, then uses the relation graph to reorder results and identify additional hits. This basic architecture is identical to that of Soules and Ganger [21].

Each content-only result is assigned its relevance score as its initial rank value. The relation graph is then traversed breadth-first for each content-only result. The *path length*, P, is the maximum number of steps taken from any starting node in the graph during this traversal. Limiting the number of steps is necessary to avoid inclusion of weak, distant relationships and to allow our algorithm to complete in a reasonable amount of time.

Further, because incorrect lightly-weighted edges may form, an edge's weight must provide some minimum support: it must make up a minimum fraction of the source's outgoing weight or the sink's incoming weight. Edges below this *weight cutoff* are pruned.

The tool runs the following algorithm, called *basic BFS*, for P iterations. Let E_m be the set of all incoming edges to node m, with $e_{nm} \in E_m$ being a given edge from n to m and $\gamma(e_{nm})$ being the fraction of the outgoing edge weight for that edge. w_{n_0} is the initial value, its content-only score, of node n. α dictates how much trust is placed in each specific weighting of an edge. At the i-th iteration of the algorithm:

$$w_{m_i} = \sum_{e_{nm} \in E_m} w_{n_{i-1}} \cdot \left[\gamma(e_{nm}) \cdot \alpha + (1 - \alpha)\right] \quad \text{(1a)}$$

After all P runs of the algorithm, the total weight of each node is:

$$w_m = \sum_{i=0}^{P} w_{m_i} \quad \text{(1b)}$$

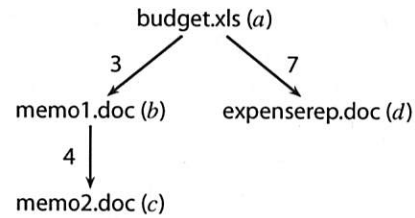


Figure 2. Relation graph used to illustrate the workings of the basic BFS algorithm.

In (1a), heavily-weighted relationships and nodes with multiple paths push more of their weight to node m. This matches user activity as files frequently used together will receive a higher rank; infrequently seen sequences will receive a lower rank. The final result list sorts by (1b) from highest to lowest value.

As an example, consider a search for "project budget requirements" that yields a content-only phase result of budget.xls with weight $w_{a_0} = 1.0$. Assume that during the context-enhancing phase, with parameters P=3, α=0.75 and no weight cutoff, the relation graph shown in Figure 2 is loaded from disk. Take node expenserep.doc, abbreviated as d. The node's initial weight is $w_{d_0} = 0$ as it is absent from the content-only phase results. The algorithm proceeds as follows for P iterations:

$$\begin{aligned} w_{d_1} &= w_{a_0} \cdot \left[\gamma(e_{ad}) \cdot \alpha + (1 - \alpha)\right] && \text{by (1a)} \\ &= 1.0 \cdot \left[(7/10) \cdot 0.75 + 0.25\right] = 0.775 \\ w_{d_2} &= 0 && \text{as } w_{a_1}\text{=0} \\ w_{d_3} &= 0 && \text{as } w_{a_2}\text{=0} \end{aligned}$$

Finally, the total weight of node d is:

$$w_d = 0 + 0.775 + 0 + 0 = 0.775 \quad \text{by (1b)}$$

The final ordered result list, with terminal weights in parentheses, is: budget.xls (1.0), expenserep.doc (0.775), memo1.doc (0.475) and memo2.doc (0.475). In this example, both memo files have identical terminal weights; ties are broken arbitrarily.

Though straightforward, this breadth-first reordering and extension mechanism proves effective [21]. We are also investigating using machine learning techniques for more accurately inferring semantic order.

4 Implementation

Our implementation runs on Windows NT-based systems. We use a binary rewriting technique [11] to trace all file system and interprocess communication calls. We chose such a user space solution as it allows tracking high-level calls in the Win32 API.

When a user first logs in, our implementation instruments all running processes, interposing on our candidate set of system calls as listed in Table 1. It

File System Operations Opening and closing files (e.g., CreateFile, _lopen, _lcreat, CloseHandle); reading and writing files (e.g., ReadFile, WriteFile, ReadFileEx, ...); moving, copying, and unlinking files (e.g., MoveFile, CopyFile, DeleteFile, ...).

IPC Operations Clipboard (DDE), mailslots, named pipes.

Other Process creation and destruction: CreateProcess, ExitProcess.

Not interposed Sockets, data copy (i.e., WM_COPYDATA messages), file mapping (a.k.a. shared memory), Microsoft RPC, COM.

Table 1. System calls which our tool interposes on. We trace both the ANSI and Unicode versions of these calls.

also hooks the CreateProcess call, which will instrument any subsequently launched executables. Care was required to not falsely trip anti-spyware tools. Each instrumented process reports its system call behavior to our background collection daemon, which uses idle CPU seconds, via the mailslots IPC mechanism. For performance reasons, each process amortizes 32K or 30 seconds worth of events across a single message. The collection daemon contemporaneously creates two relation graphs: one using temporal locality (§3.1.1) and one using causality (§3.1.2).

If a file is deleted, its node in the relation graph becomes a zombie: it relinquishes its name but maintains its current weight. The basic BFS algorithm uses a zombie's weight in its calculations, but a zombie can never be returned in the search result list. We currently do not prune zombies from the relation graph.

Content indexing is done using Google Desktop Search (GDS) with its exposed API. We expect GDS to use state-of-the-art information retrieval techniques to conduct its searches. We chose GDS over other content indexing tools, such as Indri [1], because of its support for more file types. All queries enter through our interface: only GDS's indexing component remains active, its search interface is turned off. GDS also indexes email and web pages, but we prune these from the result set. In the future, we intend to examine email and web work habits and metadata to further enhance search.

A complication arises, however. GDS allows sorting by relevance, but it does not expose the actual relevance scores. These are necessary as they form the initial values of the basic BFS algorithm (§3.2). We use:

$$\psi(i) = \frac{2(n-i)}{n(n+1)} \qquad (2)$$

to seed the initial values of the algorithm. Here, n is the total number of results for a query, and i is the result's position in the result list. Equation (2) is a strict linear progression with relevance values constrained such that the sum of the values is unity, roughly matching the results one would expect from a TF/IDF-type system [3]. Soules [20] found that equation (2) performs nearly as well as real relevance scores: (2) produces a 10% improvement across all recall levels in Soules's study, while real relevance scores produce a 15% improvement.

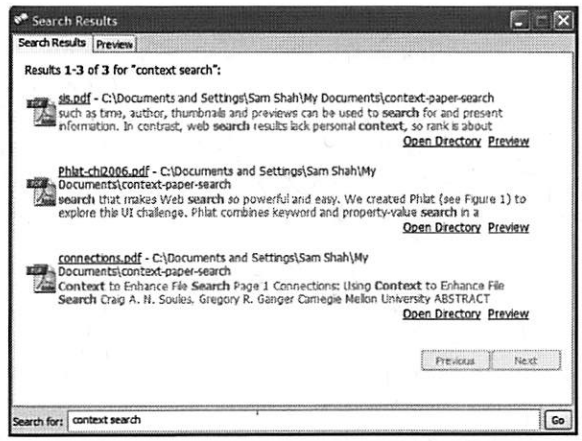


Figure 3. A screenshot of the search interface.

Users interact with our search system through an icon in the system tray. When conducting a search, a frame, shown in Figure 3, appears, allowing the user to specify her query keywords in a small text box. Search results in batches of ten appear in the upper part of the frame. A snippet of each search result, if available, is presented, along with options to preview the item before opening. Previewing is supported by accessing that file type's ActiveX control, as is done in web browsers.

In most desktop search applications, ours included, the search system is available to users immediately after installation. Because the content indexer works during idle time and little to no activity state has been captured to build our relation graph when first installed, search results during this initial indexing period are usually quite hapless. We warn users that during this initial indexing period that their search results are incomplete.

Our implementation uses a relation window of 30 seconds and basic BFS with a weight cutoff of 0.1% and parameters $P = 3$ and $\alpha = 0.75$. These parameters were validated by Soules and Ganger [21].

To prevent excessively long search times, we restrict the context-enhancing phase to 5 seconds and return intermediate results from basic BFS. Although, as shown in our evaluation (§5.3.3), we rarely hit this limit. Due to our unoptimized implementation, we expect a commercial implementation to perform slightly better than our results would suggest.

5 User Study/Evaluation

Our evaluation has four parts: first, we explain the importance of conducting a user study as our primary method

of evaluation. Second, we describe a controlled trial coupled with a rating task to assess user satisfaction. The results indicate that our causality algorithm is indeed an improvement over content-only indexing, while temporal locality is statistically indistinguishable. Third, we evaluate the time and space overheads of our causality algorithm, finding that both are reasonable. Fourth, we dissect user elicited feedback of our tool.

5.1 Experimental Approach

Traditional search tools use a corpus of data where queries are generated and oracle result sets are constructed by experts [3]. Two metrics, precision (minimizing false positives) and recall (minimizing false negatives) are then applied against this oracle set for evaluation.

Personal file search systems, however, are extremely difficult to study in the laboratory for a variety of reasons. First, as these systems exercise a user's own content, there is only one oracle: that particular user. All aspects of the experiment, including query generation and result set evaluation, must be completed by the user with their own files. Second, a user's search history and corpus is private. Since the experimenter lacks knowledge of each user's data, it's nearly impossible to create a generic set of tasks that each user could perform. Third, studying context-enhanced search is further complicated by the need to capture a user's activity state for a significant length of time, usually a month or more, to develop our dynamic indices—an impractical feat for an in-lab experiment.

In lieu of these difficulties with in-lab evaluation, Soules and Ganger [21] constructed a corpus of data by tracing six users for a period of six months. At the conclusion of their study, participants were asked to submit queries and to form an oracle set of results for those queries. Since each user must act as an oracle for their system, they are loathe to examine every file on their machine to build this oracle. Instead, results from different search techniques were combined to build a good set of candidates, a technique known as pooling [3]. Each search system can then be compared against each oracle set using precision and recall.

While an excellent initial evaluation, such a scheme may exhibit observational bias: users will likely alter their behavior knowing their work habits are being recorded. For instance, a user may be less inclined to use her system as she normally would for she may wish to conceal the presence of some files. It is quite tough to find users who would be willing to compromise their privacy by sharing their activity and query history in such a manner.

Further, to generate an oracle set using pooling, we need a means to navigate the result space beyond that returned from content-only search. That is, we need to use results from contextual indexing tools to generate the additional pooled results. However, the lack of availability of alternative contextual indexing tools means that pooling may be biased toward the contextual search tool under evaluation, as that tool is the only one generating the extra pooled results.

We also care to evaluate the utility of our tool beyond the metrics of precision and recall. Precision and recall fail to gauge the differences in orderings between sets of results. That is, two identical sets of results presented in different order will likely be qualitatively very different. Further, while large gains in mean average precision are detectable to the user, nominal improvements remain inconclusive [2]. We would like a more robust measurement that evaluates a user's perception of search quality.

For these reasons, we conduct a user study and deploy an actual tool participants can use. First, we run a *pre-post measures randomized controlled trial* to ascertain if users perceive end-to-end differences between content-only search and our causality algorithm with basic BFS. Second, we conduct a *repeated measures experiment* to qualitatively measure search quality: we ask users to rate search orderings of their previously executed queries constructed by content-only search and of results from our different dynamic techniques.

5.1.1 Background

We present a terse primer here on the two techniques we use in our user study. For more information on these methods, the interested reader should consult Bernard [4] or Krathwohl [14].

A pre-post measures randomized controlled trial is a study in which individuals are allocated at random to receive one of several interventions. One of these interventions is the standard of comparison, known as the "control," the other interventions are referred to as "treatments." Measurements are taken at the beginning of the study, the pre-measure, and at the end, the post-measure. Any change between the treatments, accounting for the control, can be inferred as a product of the treatment. In this setup, the control group handles threats to validity; that any exhibited change is caused by some other event than the treatment. For instance, administering a treatment can produce a psychological effect in the subject where the act of participation in the study results in the illusion that the treatment is better. This is known as the placebo effect.

Consider that we have a new CPU scheduling algorithm that makes interactive applications feel more responsive and we wish to gauge any user-perceived difference in performance against the standard scheduler. To accomplish this, we segment our population randomly into two groups, one which uses the standard scheduler,

the control group, and the other receives our improved scheduler, the treatment group. Neither group knows which one they belong to. At the beginning of the study, the pre-measure, we ask users to estimate the responsiveness of their applications with a questionnaire. It's traditional to use a Likert scale in which respondents specify their level of agreement to a given statement. The number of points on an n-point Likert scale corresponds to an integer level of measurement, where 1 to n represents the lowest to highest rating. At the end of the study, the post-measure, we repeat the same questionnaire. If the pre- and post-measures in the treatment group are statistically different than the pre- and post-measures in the control group, we can conclude our new scheduler algorithm is rated better by users.

Sometimes it is necessary or useful to take more than one observation on a subject, either over time or over many treatments if the treatments can be applied independently. This is known as a repeated measures experiment. In our scheduler example, we may wish to first survey our subject, randomly select an algorithm to use and have the subject run the algorithm for some time period. We can then survey our subject again and repeat. In this case, we have more than one observation on a subject, with each subject acting as its own control.

Traditionally, one uses ANOVA to test the statistical significance of hypotheses among two or more means without increasing the α (false positive) error rate that occurs with using multiple t-tests. With repeated measures data, care is required as the residuals aren't uniform across the subjects: some subjects will show more variation on some measurements than on others. Since we generally regard the subjects in the study as a random sample from the population at large and we wish to model the variation induced in the response by these different subjects, we make the subjects a random effect. An ANOVA model with both fixed and random effects is called a mixed-effects model [19].

5.1.2 Randomized Controlled Trial

In our study, we randomly segment the population into a control group, whose searches return content-only results, and a treatment group, whose searches return results reordered and extended by basic BFS using a relation graph made with the causality algorithm.

To reduce observational bias and protect privacy, our tool doesn't track a user's history, corpus, or queries, instead reporting aggregate data only. During recruitment, upon installation, and when performing queries, we specifically state to users that no personal data is shared during our experiment. We hope this frees participants to use their machines normally and issue queries without hindrance.

The interface of both systems is identical. To prevent the inefficiency of our unoptimized context-enhancing implementation from unduly influencing the treatment group, both groups run our extended search, but the control group throws away those results and uses content-only results exclusively.

The experiment is double-blind: neither the participants nor the researchers knew who belonged to which group. This was necessary to minimize the observer-expectancy effect; that unconscious bias on the part of the researchers may appear during any potential support queries, questions, or follow ups. The blinding process was computer controlled.

Evaluation is based on pre- and post-measure questionnaires where participants are asked to report on their behavior using 5-point Likert scale questions. For example, "When I need to find a file, it is easy for me to do so quickly." Differences in the pre- and post-measures against the control group indicate the overall effect our causality algorithm has in helping users find their files. We also ask several additional questions during the pre-survey portion to understand the demographics of our population and during the post-survey to elicit user feedback on our tool.

We pre-test each survey instrument on a small sample of a half-dozen potential users who are then excluded from participating in our study. We encourage each pre-tester to ask questions and utilize "think-alouds," where the participant narrates her thought process as she's taking the survey. Pre-testing is extremely crucial as it weeds out poorly worded, ambiguous, or overly technical elements from surveys. For example, the first iteration of our survey contained the question, "I often spend a non-trivial amount of time looking for a file on my computer." Here, the word "non-trivial" is not only equivocal, it is confusing. A more understandable question would be to set an exact time span: "I often spend 2 minutes or more a day looking for a file on my computer."

We also conducted a pilot study with a small purposive sample of colleagues who have trouble finding their files. This allowed us to vet our tool and receive feedback on our study design. Naturally, we exclude these individuals and this data from our overall study.

5.1.3 Rating Task

We wish to evaluate the n different dynamic algorithms against each other. Segmenting the study population into n randomized groups can make finding and managing a large enough sample difficult. More importantly, as we will show, controlled experiments on broad measurements for personal search behavior are statistically indistinguishable between groups; we believe users have difficultly judging subtle differences in search systems.

To that end, we also perform a repeated measures experiment. As we can safely run each algorithm independently, we contemporaneously construct relation graphs using both the temporal locality and causality algorithms

in both groups. At the conclusion of the study, we choose up to k queries at random that were previously *successfully* executed by the user and re-execute them. Different views, in random order, showing each different algorithm's results are presented; the user rates each of them independently using a 5-point Likert scale. We use these ratings to determine user-perceived differences in each search algorithm.

We define "successfully executed" to be queries where the user selected at least one result after execution. To prevent users from rating identical, singular result lists—which would give us no information—we further limit the list of successful queries by only considering queries where at least one pair of algorithms differs in their orderings. With this additional constraint, we exclude an additional 2 queries from being rated.

The rating task occurs at the end of the study and not immediately after a query as we eschew increasing the cognitive burden users experience when searching. If users knew they had to perform a task after each search, they might avoid searches because they anticipate that additional task. Worse, they might perfunctorily complete the task if they are busy. In a longer study, it would be beneficial to perform this rating task at periodic intervals to prevent a disconnect with the query's previous intent in the user's mind. Previous work has shown a precipitous drop in a user's ability to recall computing events after one month [6].

Finally, we re-execute each query rather than present results using algorithm state from when the query was first executed. The user's contextual state will likely be disparate between when the query was executed and at the time of the experiment; any previous results could be invalid and may potentially cause confusion.

In our experiment, we chose k=7 queries to be rated by the user. We anecdotally found this to provide a reasonable number of data points without incurring user fatigue. Four algorithms were evaluated: content-only, causality, temporal locality and a "*random*-ranking" algorithm, which consists of randomizing the top 20 results of the content-only method.

5.2 Experimental Results

Our study ran during June and July 2006, starting with 75 participants, all undergraduate or graduate students at the University of Michigan, recruited from non-computer science fields. Each participant was required to run our software for at least 30 days, a period allowing a reasonable amount of activity to be observed while still maintaining a low participant attrition rate. Of the initial 75 participants, 27 (36%), consisting of 15 men and 12 women, completed the full study. This is more than four times the number of Soules and Ganger [21]. Those who successfully completed the study received modest compensation.

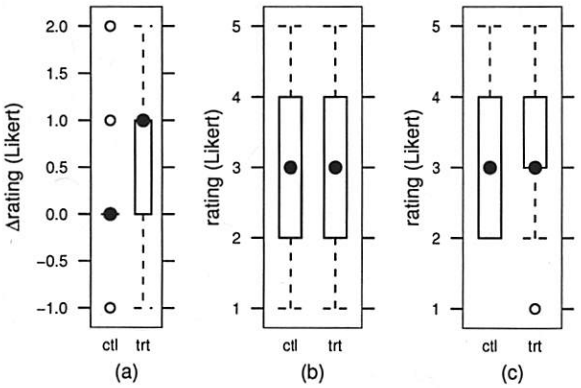


Figure 4. Box-and-whisker plot comparison between control and treatment groups on: (a) *Difference* between pre- and post-measures 5-point Likert rating of "When I need to find a file, it is easy for me to do so quickly." While the treatment group has a slightly higher median difference, the results are statistically indistinguishable. (b) 5-point Likert rating of "I would likely put less effort in organizing my files if I had this tool available." (c) 5-point Likert rating of "This tool should be essential for any computer." ($N = 27$)

To prevent cheating, our system tracks its installation, regularly reporting if it's operational. We are confident that we identify users who attempt to run our tool for shorter than the requisite 30 days. Further, to prevent users from creating multiple identities, participants must supply their institutional identification number to be compensated. In all, we excluded 4 users from the initial 75 because of cheating.

5.2.1 Randomized Controlled Trial

Evaluating end-to-end effects, as in our controlled trial, yields inconclusive results. Figure 4 shows box-and-whisker plots of 5-point Likert ratings for key survey questions delineated by control and treatment group. For those unfamiliar: on a box-and-whiskers plot, the median for each dataset is indicated by the center dot, the first and third quartiles, the 25th and 75th percentiles respectively—the middle of the data—are represented by the box. The lines extending from the box, known as the whisker, represent 1.5 times this interquartile range and any points beyond the whisker represent outliers. The box-and-whiskers plot is a convenient method to show not only the location of responses, but also their variability.

Figure 4(a) is the pre- and post-measures difference on a Likert rating on search behavior: "When I need to find a file, it is easy for me to do so quickly." Sub figures (b) and (c) are post-survey questions on if the tool

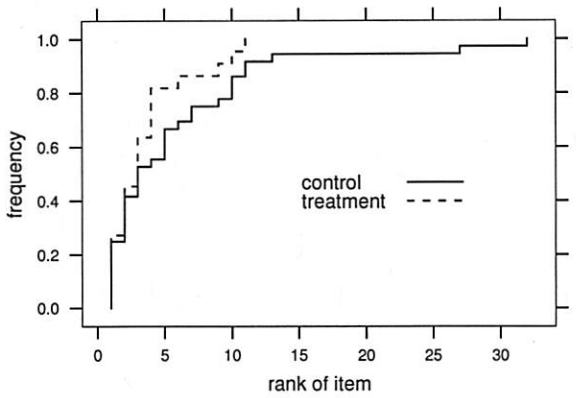


Figure 5. c.d.f. of the rank of files opened by users after a search.

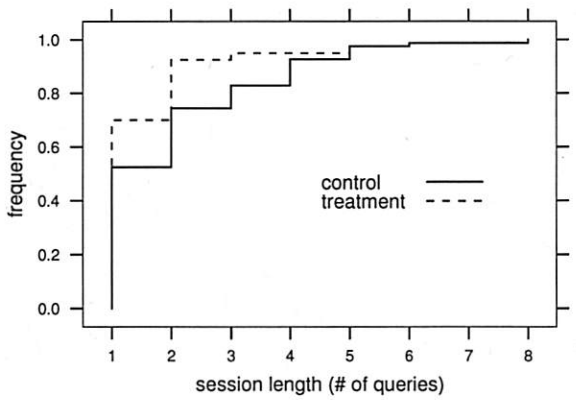


Figure 6. Session length c.d.f.

would change their behavior in organizing their files (i.e., "I would likely put less effort in organizing my files if I had this tool available") or whether this tool should be bundled as part of every machine (i.e., "this tool should be essential for any computer"). With all measures, the results are statistically insignificant between the control and treatment groups ($t_{25} = -0.2876$, p=0.776; $t_{25} = 0.0123$, p=0.9903; $t_{25} = -0.4995$, p=0.621, respectively).

We also consider search behavior between the groups. Figure 5 shows the rank of the file selected after performing a query. Those in the treatment group select items higher in the list than those in the control group, although not significantly ($t_{51} = 1.759$; p=0.0850).

We divide query execution into sessions, each session representing a series of semantically related queries. Following Cutrell et al. [5], we define a session to comprise queries that have an inter-arrival rate of less than 5 minutes. The session length is the number of queries in a session, or, alternatively, the query retry rate. As Figure 6 shows, the treatment group has a shorter average session length ($t_{97} = 2.136$, p=0.042), with geometric

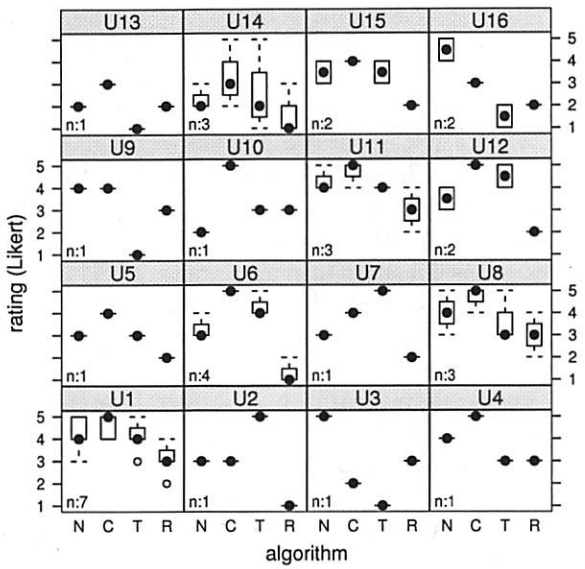


Figure 7. Box-and-whisker plots of each algorithm's ratings, delineated by subject. The algorithms are: content-only ("N"), causality ("C"), temporal locality ("T"), and random ("R").

mean session lengths of 1.30 versus 1.66 queries per session, respectively. 13.5% and 19.0% of sessions in the control and treatment groups, respectively, ended with a user opening or previewing an item.

This data is, however, inconclusive. While at first blush it may appear that with the causality algorithm users are selecting higher ranked items and performing fewer queries for the same informational need, it could be just as well that users give up earlier. That is, perhaps users fail to select lower ranked items in the treatment group because those items are irrelevant. Perhaps users in the treatment group fail to find what they're looking for and cease retrying, leading to a shorter session length. In hindsight, it would have been beneficial to ask users if their query was successful when the search window was closed. If we had such data available, we could ascertain whether shorter session lengths and opening higher ranked items were a product of finding your data faster or of giving up faster.

The lack of statistically significant end-to-end effects stems from the relatively low sample size coupled with the heterogeneity of our participants. To achieve statistical significance, our study would require over 300 participants to afford the standard type II error of $\beta = 0.2$ (power t-test, $\Delta = 0.2$, $\sigma = 0.877$, $\alpha = 0.05$). Attaining such a high level of replication is prohibitively expensive given our resources. Instead, our evaluation focuses on our rating task.

5.2.2 Rating Task

The rating task yielded more conclusive results. 16 out of our 27 participants rated an aggregate total of 34 queries,

an average of 2.13 queries per subject (σ=1.63). These 34 rated queries likely represent a better candidate selection of queries due to our "successfully executed" precondition (§5.1.3): we only ask users to rate queries where they selected at least one item from the result set for that search. 11 participants failed to rate any queries: 3 users failed to issue any, the remaining 8 failed to select at least one item from one of their searches.

Those remaining 8 issued an average of 1.41 queries (σ=2.48), well below the sample average of 6.74 queries (σ=6.91). These likely represent failed searches, but it is possible that users employ search results in other ways. For example, the preview of the item might have been sufficient to solve the user's information need or the user's interest may have been in the file's path. Of those queries issued by the remaining 8, users previewed at least one item 17% of the time but never opened the file's containing directory through our interface. To confirm our suspicions about failed search behavior, again it would have been beneficial to ask users as to whether their search was successful.

Figure 7 shows a box-and-whiskers plot of each subject's ratings for each of the different algorithms. Subjects who rated no queries are omitted from the plot for brevity. Some cursory observations across all subjects are that the causality algorithm usually performs at or above content-only, with the exception of subjects U3 and U16. Temporal locality is on par or better than content-only for half of the subjects, but is rated exceptionally poorly, less than a 2, for a quarter of subjects (U3, U9, U13 and U16). Surprisingly, while the expectation is for random to be exceedingly poor, it is often only rated a notch below other algorithms.

Rigorous evaluation requires care as we have multiple observations on the same subject for different queries—a repeated measures experiment. Observations on different subjects can be treated as independent, but observations on the same subject cannot. Thus, we develop a mixed-effects ANOVA model [19] to test the statistical significance of our hypotheses.

Let y_{ijk} denote the rating of the i-th algorithm by the j-th subject for the k-th query. Our model includes three categorical predictors: the subject (16 levels), the algorithm (4 levels), and the queries (34 levels). For the subjects, there is no particular interest in these individuals; rather, the goal is to study the person-to-person variability in all persons' opinions. For each query evaluated by each subject, we wish to study the query-to-query variability within each subject's ratings. The algorithm is a fixed effect (β_i), each subject then is a random effect (ζ_j) with each query being a nested random effect (ζ_{jk}). Another way to reach the same conclusion is to note that if the experiment were repeated, the same four algorithms would be used, since they are part of the experimental design, but another random sample would yield a different set of individuals and a different set of queries executed by those individuals. Our model therefore is:

Algorithm	β_i	95% Conf. Int. Lower	Upper	p-value†
Content only	3.545	3.158	3.932	
Causality	4.133	3.746	4.520	0.0042
Temp. locality	3.368	2.982	3.755	0.3812
Random	2.280	1.893	2.667	<0.0001
σ_1	0.3829	0.1763	0.8313	
σ_2	0.4860	0.2935	0.8046	
σ	0.8149	0.7104	0.9347	

† In comparison to content-only.

Table 2. Maximum likelihood estimate of the mixed-effects model given in equation (3).

$$y_{ijk} = \beta_i + \zeta_j + \zeta_{jk} + \epsilon_{ijk} \quad (3)$$
$$\zeta_j \sim \mathcal{N}(0, {\sigma_1}^2) \quad \zeta_{jk} \sim \mathcal{N}(0, {\sigma_2}^2) \quad \epsilon_{ijk} \sim \mathcal{N}(0, \sigma^2)$$

A maximum likelihood fit of (3) is presented in Table 2. Each β_i represents the mean across the population for algorithm i. The temporal locality algorithm is statistically indistinguishable from content-only search ($t_{99} = -0.880$, p=0.3812), while the causality algorithm is rated, on average, about 17% better ($t_{99} = 2.93$, p=0.0042). Random-ranking is rated about 36% worse on average ($t_{99} = -6.304$, p<0.0001).

Why is temporal locality statistically indistinguishable from content-only? Based on informal interviews, we purport the cause of these poor ratings is temporal locality's tendency to build relationships that exhibit post-hoc errors: the fallacy of believing that temporal succession implies a causal relation.

For example, U16 was a CAD user that only worked on a handful of files for most of the tracing period (a design she was working on). The temporal locality algorithm caused these files to form supernodes in the relation graph; every other file was related to them. Under results generated by the temporal locality algorithm, each of her queries included her CAD files bubbled to the top of the results list. U9 was mostly working on his dissertation and every file, as well as some of his music, was lightly related to each other. The temporal locality algorithm created a relation graph with 21,376 links with geometric mean weight of 1.48 (σ_l=0.688); the causality algorithm, an order of magnitude fewer, with 1,345 links and a geometric mean weight of 9.79 (σ_l=1.512). In his case, it appears that the temporal algorithm naïvely creates many lightly-weighted superfluous relations compared with the causality algorithm.

A user's work habits will affect the utility of provenance analysis techniques. Temporal locality's tendency to generate large numbers of lightweight false-positive relationships can be detrimental in many cases, mak-

ing more conservative techniques such as causality more broadly applicable.

The random reordering shares equivalent precision and recall values as content-only search, but is rated about 35.7% worse on average. We expect a random ordering to do phenomenally worse, but hypothesize that personal search tools are still in their infancy. That is, attention in the research community has been placed on web search, and only recently has desktop search become a priority. There is appreciable room for improvement. It may also be that users are simply content with having their desired result on the first page and are apathetic to each result's relative ordering within that page. More work is required to understand a user's perception of search orderings.

We further analyze any interactions between other covariates such as the demographics of participants or user features (e.g., size of disk, number of files, folder depth). We find these covariates either to be statistically insignificant or to overfit our model.

5.3 Performance

Our results indicate that our causality algorithm increases user satisfaction of personal file search. However, such a system is only effective if minimum additional system resources are required for building, storing, and traversing the relation graph created by this algorithm. We eschew discussion of content indexing overheads as these are already known [16].

5.3.1 Tracing Performance

We measure the impact building the relation graph has on foreground performance with the Postmark synthetic benchmark [13]. Postmark is designed to measure file system performance in small-file Internet server applications such as email servers. It creates a large set of continually changing files, measuring the transaction rates for a workload of many small reads, writes, and deletes. While not representative of real user activity in desktop systems, Postmark represents a particularly harsh setup for our collection daemon: many read and write events to a multitude of files inside a single process. Essentially, Postmark's workload creates a densely-connected relation graph.

We run 5 trials of Postmark, with and without tracing, with 50,000 transactions and 10,000 simultaneous files on an IBM Thinkpad X24 laptop with a 1.13 GHz Pentium III-M CPU and 640 MB of RAM, a modest machine by today's standards. The results are shown in Figure 8(a). Under tracing, Postmark runs between 7.0% and 13.6% slower (95% conf. int.; t_8=7.211, p<0.001). Figure 8(b) shows a c.d.f. of Postmark's transaction times with and without tracing across a single run. There is a relatively constant attenuation under tracing, which reflects the IPC overhead of our collection daemon and the

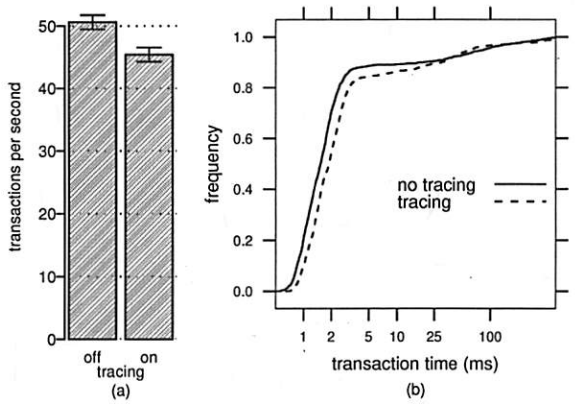


Figure 8. (a) Comparison of running 5 trials of the Postmark synthetic benchmark with and without tracing on. (b) c.d.f. of transaction times for a single Postmark run when tracing is on or off.

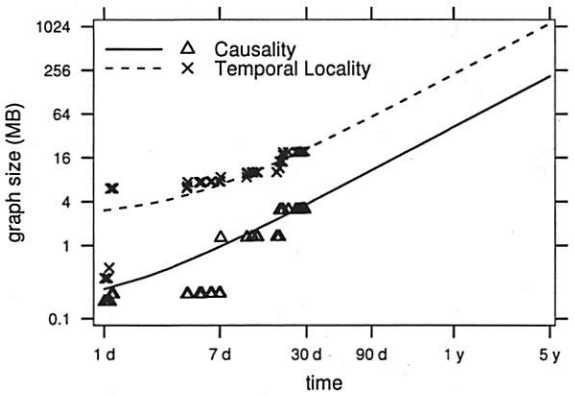


Figure 9. Relation graph growth curve for U3, the heaviest user.

additional disk utilization due to relation graph updates. This additional slowdown caused by relation graph construction is in line with other Win32 tracing and logging systems [15].

5.3.2 Space Requirements

We examine the additional space required by our relation graphs. During the user study, the tool logged the size of each relation graph every 15 minutes. Figure 9 shows relation graph growth over time for the heaviest user in our sample, U3. Each relation graph grows linearly ($r^2 = 0.861$ and $r^2 = 0.881$ for causality and temporal locality, respectively). While the worst case graph growth is $O(F^2)$, where F is the number of files on a user's system, these graphs are generally very spare: most files only have relationships to a handful of other files as a user's working set at any given time is very small. In one year, we expect the causality relation graph for U3 to grow to about 44 MB; in five years, 220 MB. This is paltry compared to the size of modern disks and represents an exceedingly small fraction of the user's working

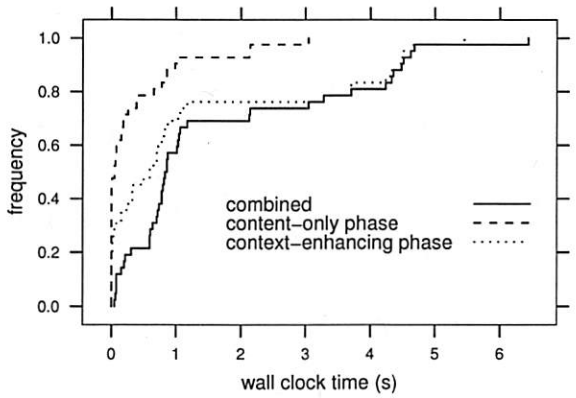


Figure 10. c.d.f. of content-only phase, context-enhancing phase, and combined wall clock times for queries issued during the user study.

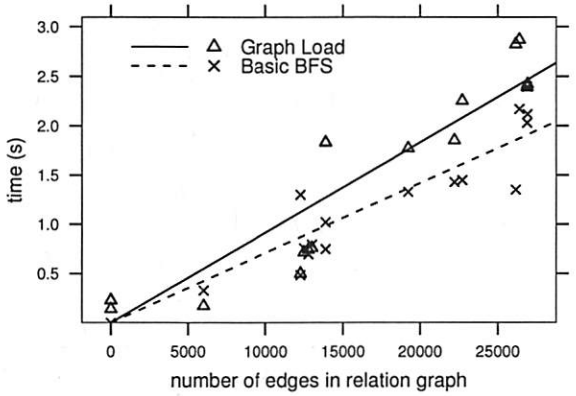


Figure 11. For queries issued during a 6-month trace of the author's system: the time spent loading the relation graph and the execution time of the basic BFS algorithm against the number of edges in the relation graph. (5 trials per query; standard deviations were within 2% of the mean for each data point.)

set. These results suggest that relation graph size isn't an obstacle.

5.3.3 Search Performance

The time to answer a query must be within reasonable bounds for users to find the system usable. In our implementation (§4), we bound the context-enhancing phase to a maximum of 5 seconds.

For every query issued during the user study, we log the elapsed wall clock time in the content-only and context-enhancing phases. Figure 10 shows these results. Half of all queries are answered within 0.8 seconds, three-quarters within 2.8 seconds, but there is a heavy tail. The context-enhancing phase takes about 67% of the entire search process on average. We believe these current search times are within acceptable limits.

Recall that the context-enhancing phase consists of two distinct subphases: first, the loading of the relation graph, followed by execution of the basic BFS algorithm (§3.2). To understand the performance impact of these subphases, previous queries issued by the author were re-executed, for 5 trials each under a cold cache, with the relation graph from a 6-month trace. Figure 11 shows the time spent for each query based on the number of edges from the relation graph loaded for that query. For non-empty graphs, loading the relation graph took, on average, between 3.6% and 49.9% longer (95% conf. int.) than the basic BFS subphase (paired $t_{15} = -2.470$, p=0.026).

Both loading the relation graph and basic BFS execution support linear increase models ($r^2 = 0.948$ and $r^2 = 0.937$, respectively). This is apparent as each subphase requires both $\Omega(F^2)$ space and time, where F is the number of files on a user's system. As these are lower bounds, the only way to save space and time would be to ignore some relationships. If we could predict a priori which relationships were most relevant, we could calculate, at the expense of accuracy, equation (1a) for those pairs. Further, we could cluster those relevant nodes together on disk, minimizing disk I/Os during graph reads.

Question	μ	σ
I would prefer an interface that shows more information.	3.84	1.34
I find it easy to think of the correct search keywords.	3.69	0.85
I would prefer if I could look over all my machines.	3.46	1.39
This tool should be essential for any computer.	3.30	1.25
I like the interface.	3.15	1.40
I would prefer if my email and web pages are included in the search results.	3.00	1.58
I would likely put less effort in organizing my files if I had this tool available.	2.92	1.12

Table 3. Additional 5-point Likert ratings asked of treatment group users at the end of the study period ($N = 13$).

5.4 User Feedback

During the post-survey phase of our study, our questionnaire contained additional 5-point Likert ratings. A tabulation of subject's responses for the treatment group are shown in Table 3. While it's difficult to draw concrete answers due to the high standard deviations, we can develop some general observations.

An area for improvement is the user interface. Our results are presented in a list view (Figure 3), but using more advanced search interfaces, such as Phlat [5], that allow filtering through contextual cues may be more useful. Different presentations, particularly timeline visualizations, such as in Lifestreams [9], may better harness users' memory for their content. There is a relatively strong positive correlation ($\rho = 0.698$) between liking

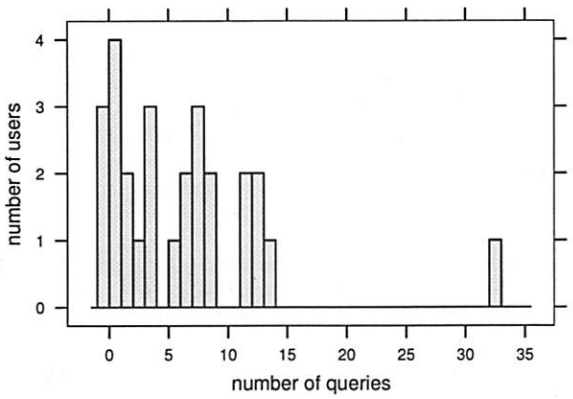


Figure 12. Distribution of the number of queries among users.

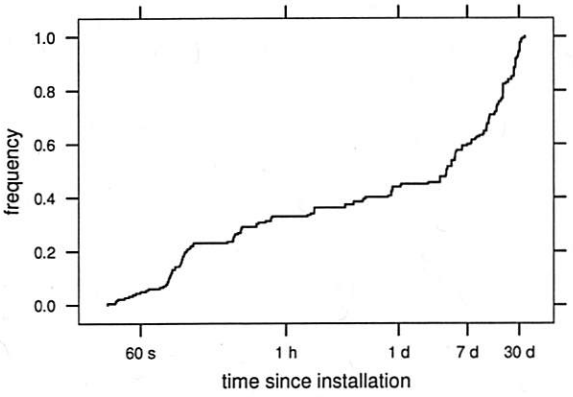


Figure 13. c.d.f. of when queries are issued after installation.

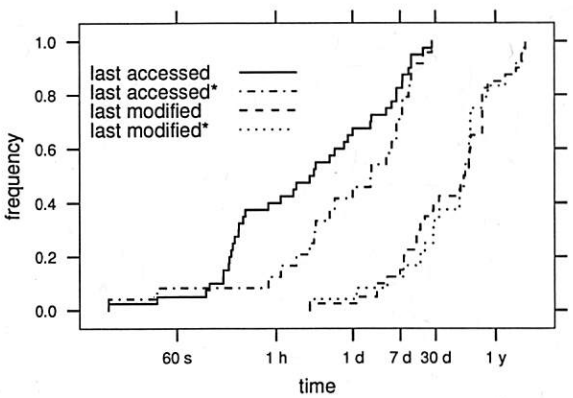


Figure 14. c.d.f. of last access and modification times of items selected from the results list. The starred versions exclude searches conducted during the first day after installation.

the interface and finding the tool essential; a better interface will likely make the tool more palatable for users.

Based on informal interviews, we found that participants used our search tool as an auxiliary method of finding content: they first look through their directory hierarchy for a particular file, switching to keyword search after a few moments of failed hunting. Participants neglect to use our search tool as a first-class mechanism for finding content. A system that is integrated into the OS, including availability from within application "open" dialogs, may cause a shift in user's attitudes toward using search to find their files.

We found it surprising that users wished to exclude email and web pages from their search results; two-thirds of users rate this question a three or below. Our consultations reveal that many of these users dislike a homogeneous list of dissimilar repositories and would rather prefer the ability to specify which repository their information need resides in. That is, a user knows if they're searching for a file, email or web page, let them easily specify which. We needn't focus on mechanisms to aggregate heterogeneous forms of context spread across different repositories into a unifying search result list, but to simply provide an easy mechanism to refine our search to a specific repository.

6 Personal Search Behavior

Finally, we explore the search behavior of our sample population. Recall that, for privacy reasons, we do not log any information about the content of users' indices or search results.

Our population issued 182 queries; the distribution per user is shown in Figure 12. The average number of queries issued per user is 6.74 (σ=6.91). Most queries, 91%, were fresh, having never been issued before. About 9% of search terms were for filenames. Since Windows XP lacks a rapid search-by-filename tool similar to UNIX's slocate, users were employing our tool to find the location of files they already knew the name of. Most queries were very short, averaging 1.16 words (σ=0.462), slightly shorter than the 1.60 and 1.59 words reported for Phlat [5] and SIS [8] respectively.

Figure 13 shows when queries are issued after installation. A sizable portion of queries are issued relatively soon after installation as users are playing with the tool. Even though we warn users that search results are initially incomplete because the content indexer has not built enough state and the relation graph is sparse (§4), it may be prudent to disallow searching until a reasonable index has been built as not to create an unfavorable initial impression.

Figure 14 shows the last access time and last modification times of items opened after searching. The starred versions represent last access and modification times of queries issued at least a day after installation. During the first day, users might be testing the tool against recent work and, hence, recently accessed files. Anecdotal evidence of this effect can be observed by the shifted last accessed curve. After the warm-up period, half of

all files selected were accessed within the past 2 days. It appears users are employing our tool to search for more than archival data.

7 Conclusions & Future Work

By measuring users perception of search quality with our rating task (§5.2.2), we were able to show that using causality (§3.1.2) as the dynamic re-indexing component increases user satisfaction in search, being rated 17% higher than content-only indexing or temporal locality, on average over all queries. While our contextual search mechanism lacked any significant increases in end-to-end effects in our randomized controlled trial (§5.2.1), this stemmed from an insufficiently large sample size. It is prohibitively expensive to secure such high levels of replication, making our rating task a more appropriate methodology for evaluating personal search systems. These results validate that using the provenance of files to reorder and extend search results is an important complement to content-only indexing for personal file search.

There is still considerable future work in this area. While we find temporal locality (§3.1.1) infelicitous in building a contextual index, one should not dismiss temporal bounds altogether. We are investigating using window focus and input flows in delineating tasks to create temporal boundaries.

Further, our tool only has limited access: a user's local file system. We could leverage electronic mail, their other devices and machines, and distributed file systems, stitching context from these stores together to provide further benefit. Since these indices may span the boundaries of multiple machines and administrative domains, we must be careful to maintain user privacy and access rights. We are investigating these and other avenues.

Finally, the tradition in the OS community, and we have been as guilty of this as any, has been to evaluate systems on a small number of users—usually departmental colleagues known to the study author. These users are generally recognized as atypical of the computing population at large: they are expert users. As the community turns its attention away from performance and toward issues of usability and manageability, we hope our work inspires the OS community to consider evaluating their systems using the rigorous techniques that have been vetted by other disciplines. User studies allow us to determine if systems designed and tested inside the laboratory are indeed applicable as we believe.

Acknowledgements

We'd like to thank Mark Ackerman for his help with our user study. This research was supported in part by the National Science Foundation under grant number CNS-0509089.

References

[1] Nasreen Abdul-Jaleel, James Allan, W. Bruce Croft, Fernando Diaz, Leah Larkey, Xiaoyan Li, Donald Metzler, Mark D. Smucker, Trevor Strohman, Howard Turtle, and Courtney Wade. UMass at TREC 2004: Notebook. In *TREC 2004*, pages 657–670, 2004.

[2] James Allan, Ben Carterette, and Joshua Lewis. When will information retrieval be "good enough"? In *SIGIR 2005*, pages 433–440, Salvador, Brazil, 2005.

[3] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, New York, 1999.

[4] H. Russell Bernard. *Social Research Methods*. Sage Publications Inc., 2000.

[5] Edward Cutrell, Daniel C. Robbins, Susan T. Dumais, and Raman Sarin. Fast, flexible filtering with *Phlat*—personal search and organization made easy. In *CHI 2006*, pages 261–270, Montréal, Québec, Canada, 2006.

[6] Mary Czerwinski and Eric Horvitz. An investigation of memory for daily computing events. In *HCI 2002*, pages 230–245, London, England, 2002.

[7] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *IMC 1999*, pages 59–70, 1999.

[8] Susan T. Dumais, Edward Cutrell, J. J. Cadiz, Gavin Jancke, Raman Sarin, and Daniel C. Robbins. Stuff I've Seen: A system for personal information retrieval and re-use. In *SIGIR 2003*, pages 72–79, Toronto, Ontario, Canada, 2003.

[9] Scott Fertig, Eric Freeman, and David Gelernter. Lifestreams: An alternative to the desktop metaphor. In *CHI 1996*, pages 410–411, Vancouver, British Columbia, Canada, April 1996.

[10] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole. Semantic file systems. In *SOSP 1991*, pages 16–25, Pacific Grove, CA, October 1991.

[11] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. In *3rd USENIX Windows NT Symposium*, pages 135–143, Seattle, WA, USA, 1999.

[12] David Huynh, David R. Karger, and Dennis Quan. Haystack: A platform for creating, organizing and visualing information using RDF. In *Semantic Web Workshop*, 2002.

[13] Jeffrey Katcher. Postmark: A new filesystem benchmark. Technical Report 3022, Network Appliance, October 1997.

[14] David R. Krathwohl. *Methods of Educational and Social Science Research: An Integrated Approach*. Waveland Inc., 2nd edition, 2004.

[15] Jacob R. Lorch and Alan Jay Smith. The VTrace tool: building a system tracer for Windows NT and Windows 2000. *MSDN Magazine*, 15(10):86–102, October 2000.

[16] Donald Metzler, Trevor Strohman, Howard Turtle, and W. Bruce Croft. Indri at TREC 2004: Terabyte Track. In *TREC 2004*, 2004.

[17] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenace-aware storage systems. In *USENIX 2006*, pages 43–56, Boston, MA, USA, 2006.

[18] Nuria Oliver, Greg Smith, Chintan Thakkar, and Arun C. Surendran. SWISH: Semantic analysis of window titles and switching history. In *IUI 2006*, pages 194–201, Sydney, Australia, 2006.

[19] José C. Pinheiro and Douglas M. Bates. *Mixed-Effects Models in S and S-Plus*. Springer, New York, 2000.

[20] Craig A. N. Soules. *Using context to assist in personal file retrieval*. PhD thesis, Carnegie Mellon University, 2006.

[21] Craig A. N. Soules and Gregory R. Ganger. Connections: using context to enhance file search. In *SOSP 2005*, pages 119–132, Brighton, UK, 2005.

[22] Jaime Teevan, Christine Alvarado, Mark S. Ackerman, and David R. Karger. The perfect search engine is not enough: a study of orienteering behavior in directed search. In *CHI 2004*, pages 415–422, 2004.

Supporting Practical Content-Addressable Caching with CZIP Compression

KyoungSoo Park Sunghwan Ihm
Princeton University

Mic Bowman
Intel Research

Vivek S. Pai
Princeton University

Abstract

Content-based naming (CBN) enables content sharing across similar files by breaking files into position-independent chunks and naming these chunks using hashes of their contents. While a number of research systems have recently used custom CBN approaches internally to good effect, there has not yet been any mechanism to use CBN in a general-purpose way. In this paper, we demonstrate a practical approach to applying CBN without requiring disruptive changes to end systems.

We develop CZIP, a CBN compression scheme which reduces data sizes by eliminating redundant chunks, compresses chunks using existing schemes, and facilitates sharing within files, across files, and across machines by explicitly exposing CBN chunk hashes. CZIP-aware caching systems can exploit the CBN information to reduce storage space, reduce bandwidth consumption, and increase performance, while content providers and middleboxes can selectively encode their most suitable content. We show that CZIP compares well to standalone compression schemes, that a CBN cache for CZIP is easily implemented, and that a CZIP-aware CDN produces significant benefits.

1 Introduction

Content-based naming (CBN) refers to a naming scheme in which pieces of content are indexed by hashes over their data. By splitting the content into smaller-sized "chunks" and obtaining their chunk hashes using a one-way cryptographic hash function (e.g., MD5, SHA-1), any content can be represented as a list of chunk hashes. The main goal behind the scheme is to reduce storage space or network bandwidth consumption by eliminating redundant chunks. Redundant chunks can be found within a single file, across files (such as snapshots of the same file over time, or collections of files), or even across machines. These latter two scenarios require building a "CBN cache," which is a cache indexed by chunk hashes.

One of the main enablers of variable-sized chunking for CBN is the Rabin fingerprinting method, which breaks a stream of data into position-independent chunks, allowing similar content to be detected even when parts of files differ [20]. A number of research systems have been developed that use CBN internally, which range from distributed file systems [2, 13, 29] and Web caching [3, 12, 21] to a cross-application transfer service architecture [28]. The commercial sector has systems which apply the same concepts to disk blocks [8], file backup [6], and WAN-link accelerators [22]. Though the use of CBN has been demonstrated in both the research and commercial sectors, there remains no easy way of applying the concept in practice without invasively changing the target platform, or designing the platform with CBN integration from the start.

Our goal in this work is to develop a file format and system that allows users to opportunistically deploy CBN while keeping their current systems intact. For example, a system employing CBN could reduce the memory footprint of Linux distribution mirrors by eliminating redundant data, since the same content is served as a DVD ISO image as well as multiple CD ISO images. At the same time, one may not want less suitable formats (unrelated RPMs or text files which rarely share common chunks among them) to be served from the CBN cache, because that would only increase the overhead with no real gain. Another similar case is transferring multiple but slightly different virtual machine (VM) images with the same base operating system from a central location (e.g., the office) to one or more destinations (e.g., home machines or an off-site facility). With typical VM image sizes ranging from many hundreds of MBs to a few GBs, placing a CBN chunk cache near the destination can help reduce a significant amount of network bandwidth by only transferring the difference after the first VM image. If the updated images are transferred in each direction when the user commutes to/from the office, the CBN can cause just the updates to be sent.

In this paper, we consider how to selectively employ CBN without requiring any support from the underlying systems. We propose a generic compression scheme based on CBN called CZIP which provides chunking, naming, and compression, allowing CZIP-aware systems to eliminate redundant chunks across files. CZIP identifies unique chunks in the input file (or stream), and then compresses the chunk by existing compression methods, such as GZIP or BZIP2. CZIP exposes chunk content hashes in the header, and CZIP-aware systems can easily recognize the content and exploit CBN caching just by reading the header information.

This approach provides an appealing alternative to designing systems around CBN, and provides some advan-

tages: (a) users or applications can better choose the set of content for CBN encoding without changing the existing environment, (b) because the file format is generic and independent of any particular system, different types of CBN caches can be utilized without sacrificing transparency, (c) even without a CBN cache, the compression scheme itself greatly reduces the content size where chunk commonality exists, and is comparable to other compression schemes in other cases.

We provide some examples of these benefits later in this paper, including the following highlights. In creating mirror servers for the Linux Fedora Core 6 distribution, CZIP reduces the data volume by a factor of 21-25 more than GZIP or BZIP2. For this kind of mirror, our server-side CBN cache provides a dramatic improvement in throughput by eliminating redundant disk reads and minimizing the memory footprint. We also integrate CBN support into the CoBlitz large-file content distribution network (CDN) [15], and show that it reduces the bandwidth consumption at the origin server by a factor of four, with no modification of the server or the client.

The rest of this paper is organized as follows: we provide some motivating examples for CBN and the CZIP format in Section 2. We then provide details about the design of CZIP in Section 3, and describe some typical deployment scenarios in Section 4. We perform some experiments on CZIP's effectiveness on different data types in Section 5, and evaluate the performance of two CZIP-augmented systems in Section 6. Finally, we discuss related work in Section 7 and then conclude.

2 Motivation

To illustrate the benefits of a common CBN-enabling format, we discuss a few candidate scenarios below. All of these examples are from systems we (and our colleagues) have built or are building at the moment, so an approach like CZIP, rather than just being theoretically interesting, actually stands to provide us with practical benefits.

2.1 Software Distribution

Software distribution over the network has been gaining popularity, especially as broadband penetration has increased, making download times more reasonable. Linux distributions are just one example of these kinds of systems, with popular projects like the Fedora Core distribution having over 100 mirror sites [1]. However, as users come to expect more capabilities, features, and packages bundled with the OS, the download sizes have increased, and the Fedora Core 6 distribution spans five CD-ROM images or one DVD-ROM image, at a total size of 3.3 GB. Since users may desire one format over another, any popular mirror site must keep both, requiring over 6 GB of space for just a single architecture. While this disk space is a trivial cost, the real problem is when this data is being served – it is larger than the physical memory of most systems, so it causes heavy disk access. The Fedora Core project has also been providing images for the 64-bit x86 architecture since the release of Fedora Core 2, and PowerPC since Fedora Core 4. While the 64-bit x86 extensions were originally available only in higher-end processors, their migration down the hierarchy to lower-end machines has also increased the demand for the x86/64 Fedora Core distribution.

Even if a mirror site provided only the two x86 distributions in both DVD and CD formats, the total size is over 12 GB. Unfortunately, this figure exceeds the physical memory size of most servers. Releases of new Fedora Core distributions tend to cause flash crowds – our own CoBlitz large-file distribution service experienced peak downloads rates of over 1.4 Gbps aggregate, and sustained rates over 1.2 Gbps [2]. In these scenarios, thousands of simultaneous users are trying to download from mirror sites, and between their sheer numbers, varying download rates, and different start times, virtually all parts of all of the files will be in demand simultaneously, causing significant memory pressure. Some popular mirror sites were unable to serve at their peak capacity due to the memory thrashing effects. One mirror site operator with 2 Gbps of bandwidth was only able to serve 500 Mbps since his system had only 2 GB of physical memory and was heavily thrashing [3].

The reason that CBN is important in these scenarios is because much redundant data exists, both across media formats (CD vs DVD) as well as across distributions for different architectures (x86 vs x86/64). The reason for the former is simple – the same files are simply being rearranged and placed on media of different sizes. The reason to expect similarity across architectures is that not all of the files in any distribution are executables. So, while the executable files may have virtually no similar chunks across different architectures, all of the support files, including documentation (PDFs, HTML pages, GIF and JPEG images, etc.) will likely be the same, as will many of the program resource files (configuration files, skins/textures, templates, sample files, etc.). We can expect savings at both of these levels, driving down the memory footprint required for serving multiple architectures. In an ideal scenario, we would expect no overhead for serving both the CD and DVD images, and each additional architecture would only expand the memory requirements by the size of the executable files.

[1] http://fedora.redhat.com/download/mirrors.html

[2] http://codeen.cs.princeton.edu/coblitz/

[3] mirror-list-d@redhat.com, Oct 26, 2006

2.2 Virtual Machine Image Mobility

Another area where large amounts of similar content are expected to occur is in the handling of virtual machine images. While virtual machines have been popular for server consolidation and management, they are also being explored for providing mobility and management. In the management scenario, VMWare has created a library of "appliances," pre-configured VM images for certain tasks [4]. Most of these images will be very similar, since they use the same base operating system.

In the mobility area, virtual machines are being used to transport user environments. Rather than having users work on laptops that they take with them, this approach relies on servers that keep a virtual machine image of the user's environment, which can be moved to whatever machine the user has available. In this way, users are not tethered to any particular physical machine even if they may use the same office and home machines repeatedly in practice. This work has been explored in the Internet Suspend/Resume (ISR) Project [25].

In such an environment, we would expect three sources of similar content – common chunks between an image and the same image at a different point in time, common chunks across images of the same or similar operating systems, and common chunks within an image. While the first two sources are easy enough to understand, the last source can arise from practices such as page-level granularity for copy-on-write – multiple instances of the same program may have only differences in their globals and heap, but these differences would have required duplicating the pages where they reside. By eliminating redundant content from all of these sources, we can expect faster time to download/upload image snapshots, as well as less memory pressure on the machine serving the images. While the ISR project has an accompanying content-addressable storage system, it uses fixed-size chunks in the range of 4-16KB [14], making it likely to only find common page-aligned content, such as executables. However, program data, etc., which may be allocated in slightly different portions of memory from image to image, may be missed even when commonality is high. Using a more sophisticated CBN approach is likely to find more commonality, and to increase the benefits from redundancy elimination. We note that the ISR project intentionally chose fixed-size chunks due to concerns about start-up times using variable-sized chunks. Later in the paper, we discuss what features the CZIP format has to support these kinds of environments efficiently.

[4] http://www.vmware.com/vmtn/appliances/directory/

2.3 Uncacheable Web Content

Web proxy caches have been the focus of much research, and the increasing capacities of disk and physical memory now enable proxies to store and index large data volumes, and to achieve high cache hit rates limited only by the HTTP-specified cacheability of the data stream [24]. However, even when content providers specify the data should not be cached, it may be the case that the data is slowly changing, and amenable to caching. For example, most news sites are only updated a handful of times per day, and even the updates leave most of the page the same – only a few articles on a page are likely to change during an update, with the rest of the page staying the same. Even sites with user-editable content, such as Wikipedia or bulletin boards, are unlikely to completely change from edit to edit.

In these scenarios, a CBN scheme can exploit the slowly-changing nature of the data to reduce bandwidth consumption, whereas a standard HTTP proxy would be prevented from caching the content at all. The prohibitions on caching are specified by the content providers via HTTP headers returned with the request. While providers can benefit from reduced bandwidth consumption when users cache content, enabling caching for these kinds of dynamically-generated content is problematic – providers often do not know when the next modification will occur, so allowing content to be cached would result in users seeing stale versions of the page.

A CBN-aware cache can work with an HTTP proxy to share responsibilities, since each is better suited for certain portions of the workload. For example, when an HTTP proxy is allowed to cache content, it need not contact the origin server during the caching period. Only when the content expires and a client requests it does the proxy need to contact the server to re-validate. The CBN cache, in contrast, must always contact the origin server when fetching dynamic content, but it may be able to avoid actually downloading the data if it is found to have not changed. A similar approach has been proposed at the router level [26], and works without any explicit HTTP-level cooperation. Later in the paper, we will discuss why exposing HTTP-level details can help optimize these kinds of transfers, and how an explicit CZIP-aware proxy can take advantage of the extra information not available at the router level.

3 Design & Implementation

CZIP is conceptually a very simple compression format that detects and eliminates redundant chunks in the input file or stream. It exposes each chunk's information in the header of the output file and compresses the chunk data itself using existing compression schemes such as GZIP or BZIP2. The overall format is shown in Figure 1. Be-

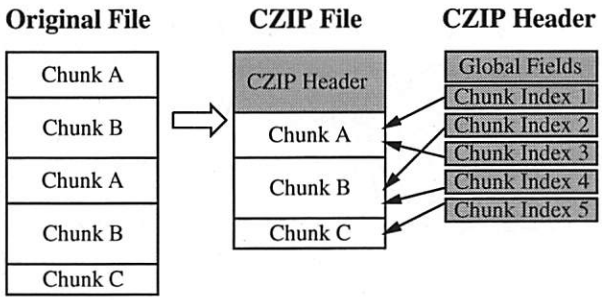


Figure 1: CZIP File format: The CZIP encoder creates the chunk index structures in the header and stores only unique chunks in the body. The stored chunk is compressed and convergently encrypted, if requested.

low, we first describe the various fields, and then provide a more detailed rationale for the design of the file format.

3.1 Header Format

The CZIP header consists of two parts – a fixed-size header that describes the overall file, and then a variable-sized header that contains the chunk information. The header is designed to be efficient and flexible, allowing applications to download what they need, and to make CZIP processing as efficient as possible. At the same time, it is designed to allow random access, even when chunk sizes are unpredictable. The specific fields of the header are:

- **Magic Cookie** – just a well-known value used to identify the CZIP format
- **Version** – which version of CZIP is used, for future expansion
- **Footer** – does this file use a footer instead of a header. If this field is set, the chunk array and real header occurs at the end of the file, not the head. The rationale is described below.
- **Sizes** – sizes of original data, total CZIP file size, and total header size.
- **Hash Format** – which hash function is being used, such as MD5, SHA-1, etc.
- **Compression Format** – which compression scheme is used, such as GZIP, BZIP2, etc.
- **Chunk Hash Size** – size of hash values in chunk array and file hashes, used to calculate positions in chunk hash array
- **Encrypt** – encryption schemes used for convergent encryption, such as DES, AES, etc. (see below)
- **Num Chunks** – number of chunks in the file
- **Header CRC** – CRC value over total header, used to detect corruption
- **File Hashes** – hash value over original and CZIP files, used to detect corruption
- **Hash Array Pointer** – the offset of the start of the chunk hash array, mostly used when the file has a footer instead of a header. Otherwise, finding the start of the hash array would be problematic.

This fixed-size header is followed by an array of per-chunk headers. These contain information about each chunk, and are designed to allow easy access.

- **Chunk Sizes** – original and compressed size of the chunk
- **Offsets** – locations of this chunk in the original and CZIP formats
- **Hash Values** – hash values for the original and processed data in this chunk

3.2 Format Rationale

The CZIP format is designed to be easily usable at several levels – including applications that need to just get general data about the file, applications that need to process the entire file, and applications that need to randomly access the file. It is also designed to be relatively easy to generate, given the constraints inherent in compressing a file using CBN. Some of the considerations involved are described below:

Planning Tools – Some tools may not care about the exact data in a CZIP file, but may be interested in knowing how much space the decompressed file requires. These tools could read just the fixed-size header to get this information.

Random Access – If a CZIP file is being used to represent a file with sparse access, programs need only read the full header to get the array of chunk headers. Since the offsets in the original file are recorded, a binary search of the chunk header array can be performed quickly, without the need to calculate offsets by adding all preceding chunk sizes.

Streamability – For most applications, having the file summary and chunk array information at the head of the file is the most useful. However, when the CZIP file is to be streamed as it is created, having all of the data in the header would require a full pass over the file, which would require buffering/creating the entire CZIP file before sending it. In this scenario, the "Footer" flag can be set, and any filled values in the file's header are viewed as advisory in nature. When a footer is used, the arrangement of the CZIP file is as follows: advisory header,

chunks, chunk hash array, file hashes, fixed-size header. In this manner, the fixed-sized header can still be found quickly when the file is retrieved from storage.

Creation Flexibility – If a variable-sized chunking scheme is used, the exact number of chunks may not be known in advance, but an approximate number can be used. If the file is being written to disk as it is created, space can be left after the chunk hash array to allow some extra space beyond the expected number of chunks. Since the per-chunk information specifies offset in the file, actual chunk data does not need to immediately follow the end of the chunk hash array. Otherwise, the creation process would have to create the header file and compress chunks separately, and then merge them. By allowing extra space between the array and the start of the chunks, the output file only has to be created once. Obviously, if footers are being used, this approach is not needed.

Encryption – A convergent encryption scheme (using DES or AES) may be applied to each chunk. Convergent encryption encrypts each chunk with its content hash as an encryption key so that the encoded chunk can be shared among authorized users [7]. The keys, which are the original content hashes, are again encrypted by a public cryptographic algorithm (such as RSA) and delivered to the authorized users. The default methods in CZIP are SHA-1 for hashing, GZIP for compression, and no encryption for the chunk content.

Integrity – The CZIP format has several mechanisms for integrity. The chunk content hash is calculated after compression and encryption are applied to the original content. This allows applications to check the integrity of a CZIP file without decompressing it.

3.3 Chunking Specifics

CZIP supports two chunking methods: fixed and variable-sized chunking. Fixed-sized chunking is simplest but if an update causes content to get shifted slightly, all previously-detected chunks after the modification point would become useless. To address the problem, the Rabin fingerprinting method is often used. Rabin's fingerprints use a random polynomial called a Rabin function with n consecutive bytes as input. A chunk boundary is determined when the function's output value modulo *average chunk size*, M, is equal to a predefined value, K (K is an integer, $0 \leq K < M$). Say M is 32 KB, and K is 17. Because the output values modulo M are well distributed over [0..2^{15}-1], the probability of the output value being 17 is close to 2^{-15}, which means the chunk boundary is formed every 32 KB on average when the Rabin function is evaluated at each byte. One advantage of Rabin's fingerprints is that even if the content is modified, that does not affect the chunking boundary beyond the modified chunk and its neighbors. Thus, most of the previously detected chunks can be reused regardless of local updates. One drawback is that the chunk size is variable, making it harder to know in advance exactly how many chunks a given piece of content produces.

By default, CZIP uses Rabin fingerprinting with a 32 KB average chunk size and GZIP compression, but these parameters can be adjusted by command-line options. For most of our workloads, 32 KB is small enough to expose most chunk commonalities and big enough to fully utilize the network socket buffers. GZIP, the default chunk compression scheme used by CZIP, finds redundant strings within a 32 KB sliding window [16], so a larger chunk size may not produce significant extra compression from GZIP.

4 Deployment Options

The overall goal of CZIP is to recognize the value of CBN by proposing an easily-handled format that provides a migration path from current compression schemes to content-based naming without requiring invasive changes or significant system redesign. While basic support for the CZIP format provides its own benefits, the design of CZIP can easily enable other benefits when used with infrastructure that is CBN-aware. In this section, we describe some deployment scenarios to maximize the benefits from CZIP. We examine what can be obtained with a CZIP-aware server, a CZIP-aware client, or both.

In this discussion, we focus on deployment using HTTP, but any appropriate protocol could be used. In particular, FTP and RSYNC servers would also be good candidates for CZIP support. Our focus on HTTP is due to its widespread adoption, and our assumption that more people know its protocol details than other protocols.

4.1 CZIP-Aware Server

A CZIP-aware server makes more efficient use of its memory when serving high-similarity content by using a CBN-based cache to reduce its working set size. In this scenario, shown in Figure 2, content providers distribute similar files encoded using CZIP, and their clients decompress the downloaded files like they would handle any other compressed formats. The CZIP files received by the client are fully self-contained. However, on the server, a chunk cache is maintained that is used across files, reducing the working set size when similar files are being served. Whenever a CZIP-file request is received, the server reads the chunk index structures from the requested file's header, and checks to see if the chunks are already loaded in the CBN cache. Any cache misses are served from the requested file, with the CBN cache also receiving the data. In this manner, the server avoids polluting main memory with redundant data, and

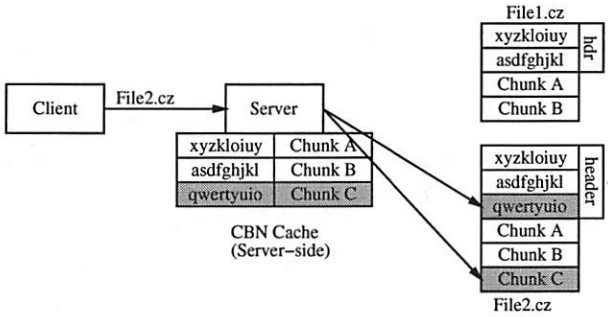


Figure 2: A CZIP-aware server needs to read only the chunk index structures and then the missing chunk C to serve File2.cz to the client. Chunk A and B are already in the server's CBN cache from a previous fetch of File1.cz.

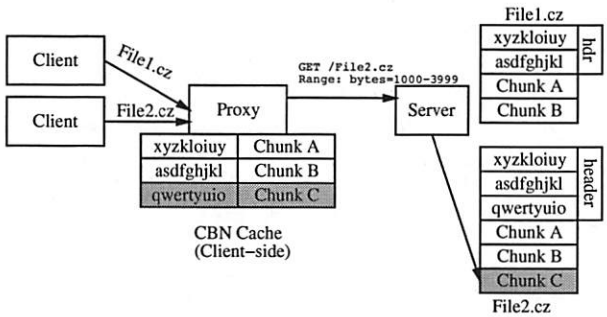


Figure 3: A CZIP-aware proxy first fetches the header of File2.cz, and downloads only chunk C. The chunk request includes the chunk's byte-range and its content hash. We assume that another client previously downloaded File1.cz.

the server's effective working set size is the union of all the unique chunks in the CZIP'ed files being served.

Since the data stream received from the server is just another file, no changes are required on any intermediary devices or at the client in order to download the CZIP file. Obviously, the client must be able to uncompress the CZIP format, which can be achieved via stand-alone programs, as a browser plug-in, as a helper application, or even with integrated browser support, as is done with GZIP'ed objects.

If this support is not feasible at the client side, the server could un-CZIP the data as it is being sent to the client, or even re-encode it using GZIP or ZLIB/deflate depending on what the client specifies in the HTTP "Accept-Encoding" header. A clever system may be able to take GZIP-encoded chunks from the CZIP file and serve them to GZIP-capable clients without a full decompression step, but this approach requires knowing some low-level details of the ZLIB stream format, and is beyond the scope of this paper. In any case, it is easy to see that a CZIP-aware server could still obtain memory footprint benefits even with a completely unmodified client.

4.2 CZIP-Aware Client

If the server is only a standard Web server with no special support for CZIP, a CZIP-aware client can still independently exploit CZIP-encoded files using only the standard HTTP protocol. The advantage for the client is lower bandwidth consumption and faster download times. In this scenario, one or more clients maintain a CBN cache that stores recently-downloaded chunks. When clients want to download a CZIP-encoded file, they ask for only the CZIP headers using the HTTP byte-range support that has been present in Apache and IIS since 1996. The clients ask for at least as many bytes as the fixed-size header, and if the response does not contain the full chunk hash array, another request can be sent to get the rest of the variable-sized header. Note that in the unlikely event that the CZIP files are stored with footers, in streaming order, the byte-range support can also ask for bytes at the end of the file.

Once it has the header, the client knows the chunk hashes, so it can try to fetch the chunks from its local CBN cache. For any chunks it does not have, the chunk hash array also contains the byte positions of the chunks within the CZIP file, so the client can use the byte-range support to just ask for specific portions of the file containing the chunks it needs. These chunks are also inserted into the CBN cache.

Alternatively, this level of support could be added to a client-side proxy server, as shown in Figure 3 so that clients themselves do not have to be aware of the CZIP format. If the connection between the client and the proxy is faster than the speed of the wide-area network, the download will still be faster than if the client had contacted the server directly.

4.3 CZIP-Awareness at Both Endpoints

The greatest benefit using CZIP arises when both endpoints are CZIP-aware and utilize CBN caching. In this scenario, the client's first request to the server retrieves the full CZIP header for the file. After consulting with its local CBN cache to determine which chunks it does not have, the client contacts the server to request the chunks by their chunk hash information instead of requesting range requests of the CZIP-encoded file. In this manner, both the client and server are only dealing with chunks rather than files when serving the body of the request, reducing both bandwidth consumption and the server's working set. With persistent connections and request pipelining, any gaps between serving individual chunks can be minimized.

This scenario requires more infrastructural change than the two previous scenarios, but even these changes could be incorporated into proxy servers. Proxy servers are often deployed as "server accelerators" or "reverse

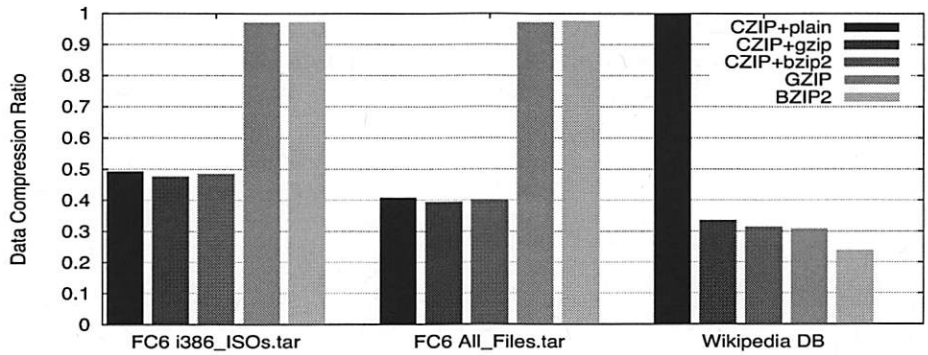


Figure 4: Data Compression Ratio: CZIP vs. GZIP and BZIP2

	Orig Size	CZIP						GZIP		BZIP2	
		Size			Time						
		plain	gzip	bzip2	plain	gzip	bzip2	Size	Time	Size	Time
FC6 (i386)	6.65	3.27	3.18	3.23	428	765	2004	6.46	846	6.47	3964
Wikipedia	7.94	7.94	2.67	2.50	656	1419	3079	2.45	976	1.90	3151
FC6 (all)	49.73	20.27	19.63	19.86	323	5194	12767	48.34	6424	48.53	29004

Table 1: Compression Performance. All sizes in GB, and all times in seconds.

proxies," where the content provider will have incoming requests pass through a proxy server before reaching the actual Web server. This approach is used to offload requests from the Web server, since the proxy may be more efficient at serving static content. In this scenario, CZIP support merely needs to be added to the proxy server, and when a CZIP-enabled client-side proxy realizes it is communicating with a CZIP-enabled server-side proxy, it can use the CZIP-aware protocols for transfers. In this manner, the changes are more localized than requiring modifications to all Web servers.

5 Compressibility Experiments

In this section, we perform a number of experiments to demonstrate the effectiveness and performance of CZIP on a range of data types, focused on the scenarios we described in Section 2. CZIP reduces data volume by first finding and eliminating redundant data, and then passing the remaining data through existing compression schemes. Not surprisingly, CZIP is most useful where a high level of chunk commonality is expected, but its compression performance does not degrade much even when there is little commonality because each chunk is individually compressed. The experiments described below compare CZIP's compression performance with GZIP and BZIP2 in terms of compression ratio and speed.

5.1 Linux Distributions

Our first experiment examines CZIP in an environment where we can expect significant data commonality, serving the Fedora Core Linux distribution [17] across its three CPU architectures (i386, x86_84 and ppc) and two media formats, DVD and CD ISOs. The byte count for just the 32-bit i386 architecture is 6.7 GB, while all Fedora Core 6 (FC6) ISOs together is about 22 GB. The full FC6 mirror, including all source RPMs, is 49.7 GB.

We prepare two sets of files, just the i386 DVD/CD ISOs, and all FC6 mirrored files, including the source RPMs. We *tar* each set into a file, and apply CZIP, GZIP and BZIP2 each on a machine with a 2.8 GHz Pentium D processor and 2 GB of memory. Each file is compressed with just CZIP alone (no chunk compression), CZIP used with GZIP or BZIP2 as a per-chunk compressor, and then GZIP and BZIP2 used alone as standalone compressors. In all cases, whether running standalone or in conjunction with CZIP, the GZIP and BZIP2 compressors use their default compression parameters.

Figure 4 shows the data compression ratios, which are calculated as compressed size/original size. More details are shown in Table 1. The results show that all of the CZIP variants, even with no chunk-level compression, yield files less than half the size of GZIP and BZIP2, which show virtually no compression. The CZIP results are not surprising, because the DVD ISO contains all the data of the CD ISOs, and the ISOs again contain all the RPM contents. The CZIP'ed-ISO file (3.17 GB) is actu-

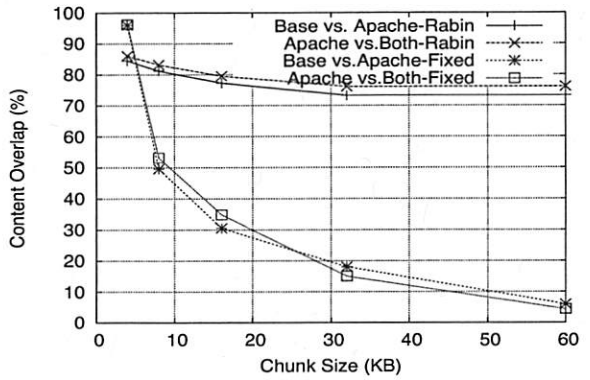


Figure 5: Content overlap over similar Xen Linux VMs

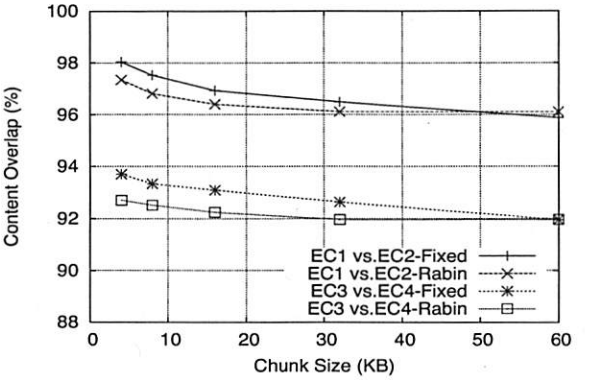


Figure 6: Content overlap over engineering computing Xen Linux VMs after three weeks of use

ally smaller than just one DVD ISO (3.28 GB) because of the compression of each chunk's data. On the other hand, GZIP and BZIP2 do not find the chunk-level commonality across the ISOs, nor can they compress the ISOs much further because most of their contents are already compressed. By the actual byte counts, CZIP saves 3.3 GB and 28.7 GB more disk space for each FC file than the other schemes. The 32KB chunk size performs well enough – dropping to 4KB using CZIP alone only compresses the file to 19.07 GB from 20.27 GB. The smaller chunk size does provide a small speed boost, reducing compression time to 2583 seconds from 3231, presumably due to processor-cache effects.

5.2 Wikipedia

Our next test involves a large set of human-generated content, an offline version of the Wikipedia [32] database containing all of its pages, which is intended for proxying in regions where bandwidth is limited. We download the latest version of the database file (produced on 11/30/2006), and tested each compression scheme on it.

The data compression ratio in Figure 4 shows that CZIP with no compression performs the worst, which is in contrast to the Fedora Core cases. The Wikipedia database file shows almost no chunk-level commonality ($< 0.001\%$) using the default average chunk size of 32 KB, while it is easily compressed with other methods. This file presents a worst case scenario for CZIP, since the chunking scheme presents smaller pieces of data for the individual compressors. The difference for CZIP+GZIP versus GZIP alone is an additional 2.8%, since GZIP uses a relatively small (32KB) window. BZIP2, however, uses a much larger compression window (900KB), and is able to gain an additional 7.6% over the combination of CZIP+BZIP2, since CZIP produces chunks smaller than BZIP2's window.

5.3 Virtual Machine Images

Handling multiple virtual machine (VM) images for the same operating system is another area where much cross-file commonality would be expected. We investigate this scenario using two sets of images – one that is server oriented, and another reflecting client machines.

The server test creates three Xen [4]-based Linux VM images: "Base," a minimum-functionality (with no redundant daemons) Fedora Core 4 image, "Apache," which adds the Apache Web server to the base image, and "Both," which adds the MySQL database server to Apache. Each image is created on a 2 GB file-based disk image, but the real content sizes, measured by unique chunks, are 734.8, 782.3 and 790.8 MB, respectively.

In Figure 5, we compare the content overlap ratio between Base and Apache, and between Apache and Both over different chunk sizes and chunking methods. We see more content overlapping with smaller chunks, because the granularity of comparison gets smaller. The performance of fixed-size chunking degrades significantly after 4 KB, the hardware page size. However, variable chunking degrades much more slowly and flattens after 16 KB. Using Rabin's fingerprinting method, we detect 73-86% of redundancy regardless of chunk sizes.

For the client test, we create five identical Xen Linux VMs (EC1-EC5) configured with a standard engineering computing (EC) for a large technology company. For three weeks, five different engineers extensively used the VMs for various tasks in the hardware design process. The image size is 4 GB each, but the real content per VM is 2.2 GB. In Figure 6, we show the overlap between two pairs of images, EC1 and EC2, and EC3 and EC4 (comparison with other pairs is similar). CZIP finds more than 90% redundancy between all pairs of VMs, and sometimes as much as 98%. Interestingly, fixed-size chunking degrades much less in this test, and even slightly outperforms Rabin fingerprinting. The reason is because most

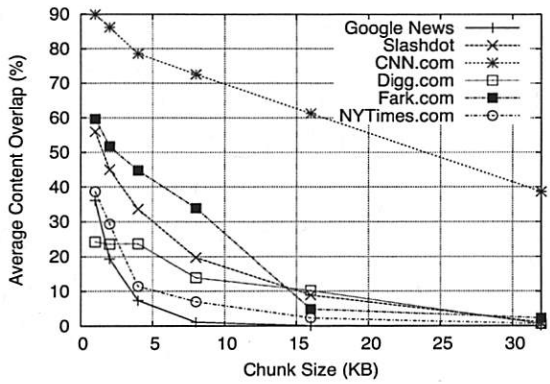


Figure 7: Average content overlap for uncacheable Web pages

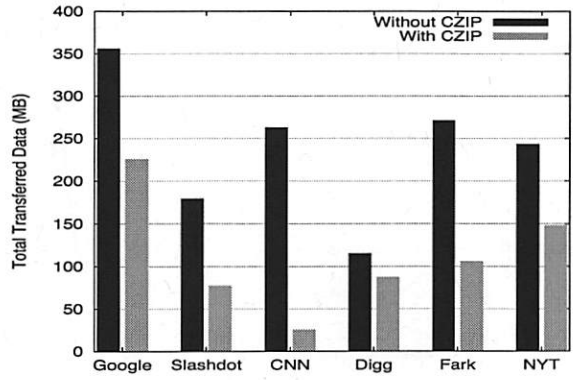


Figure 8: Potential transferred data savings using CZIP with 1-KB chunks

of the content did not change over the three weeks, and if the layout of the content is aligned with multiples of the fixed chunk size, fixed-sized chunking can find more commonality.

5.4 Dynamic Web Pages

Our final compressibility test examines the commonality between multiple snapshots of dynamic Web sites over time. We download the front pages of Google News, CNN, Slashdot, Digg.com, Fark.com and the New York Times (NYT) every 10 minutes for 18 days. All of these sites mark their front pages uncacheable with "no-cache," "no-store," or "private" in the "Cache-Control" response header, which would not only render shared HTTP proxy caches useless, but would also prevent browser caching in most of their cases. The data volumes for the HTML alone range from 120-360 MB for the entire period.

We run CZIP on each snapshot, and for each site, we compare the chunk overlaps on every pair of snapshots taken 10 minutes apart. Figure 7 shows the average content overlap during the 18-day period for each site, using CZIP runs with varying chunk sizes. As in the previous section, we see that the commonality decreases as the chunk size increases, but we see 24% to 90% average redundancy for 1-KB chunks. The particular pattern is also interesting – Google News shows the worst savings at the 4KB chunk size, since most of the blurbs are small and their positions are updated frequently. The entertainment site Fark.com uses much smaller blurbs and updates roughly 50 times per day, but shows high commonality, due to the blurbs getting added and removed from the front page in FIFO order. As such, the Rabin fingerprinting approach can still work with the shifted content. The per-site savings in bytes transferred using CZIP-aware systems is shown in Figure 8.

	CZIP			GZIP	BZIP2
	plain	gzip	bzip2		
FC6 (i386)	130	169	1299	229	2021
Wikipedia	146	200	786	218	1167
FC6 (all)	1465	1865	10009	1708	14971

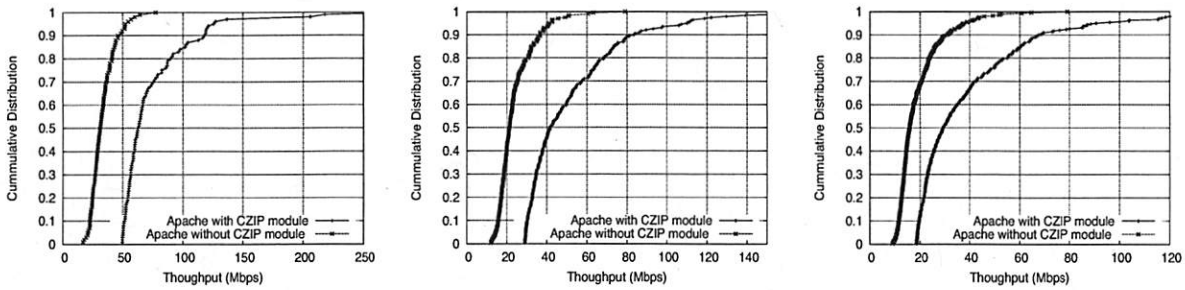
Table 2: Decompression Performance. All times in seconds.

5.5 Overheads

CZIP's compression and decompression speed are mostly comparable to GZIP and much better than BZIP2. Tables 1 and 2 show the times taken for compressing and decompressing FC6 and Wikipedia DB files. For the FC6 files, CZIP finishes 91 seconds (10.6%) and 1229 seconds (19.1%) earlier than GZIP in compression because it can avoid processing redundant chunks. But CZIP is 443 seconds (45.3%) slower than GZIP for the Wikipedia DB file. This is due to CZIP's chunking overhead and redundant file access for temporarily saving intermediate chunks before writing the header. Decompression is generally much faster than compression, and its performance is usually bounded by the disk write speed. CZIP's decompression speed is comparable to that of GZIP. In comparison with BZIP2, CZIP is 2.2 to 5.6 times faster in compression and 5.8 to 12 times faster in decompression, mostly due to BZIP2's CPU-heavy reconstruction process during decompression.

6 Performance Evaluation

In this section, we evaluate the performance benefits of CZIP-aware systems in two contexts – a server-side CBN cache, and CZIP integration with a content distribution network (CDN). The CBN cache is implemented as a module on the Apache Web Server, and the CDN support is integrated into the CoBlitz large-file CDN [15].



(a) 100 clients (b) 200 clients (c) 300 clients

Figure 9: Client throughput distribution when downloading large ISO image files

6.1 Server-Side CBN Cache

Our server-side CBN cache is evaluated for one of the deployment scenarios that we described earlier in Section 2 – a software distribution mirror handling a data set larger than its physical memory. In this scenario, the server can easily experience thrashing, and have its throughput bottlenecked by disk access performance. A server-side CBN cache can help avoid unnecessary disk reads and reduce the effective memory footprint of the server.

Our implementation consists of an Apache module which handles a CZIP file request by parsing the file's header and fetching the chunks from a CBN cache server. The CBN cache server is a user-level file chunk server on the same machine that caches chunks indexed by CBN. The module sends a chunk request with a file path, a byte-range and a chunk content hash. The CBN cache server finds the chunk in its cache or reads it from the file system on cache misses. It can be configured to recheck the chunk content hash for possible attacks or corruption. Because the CBN cache server is a separate process, any CZIP-aware servers on the same machine can benefit from the cache as well.

We use a server machine with a 2.8 GHz Pentium D processor, 2 GB memory, and two Gigabit Ethernet network interface cards (NICs). We compare Apache 1.3.37 with and without the CZIP module on a data set consisting of a 1.5 GB file extracted from the Fedora Core 6 DVD ISO and three 0.5 GB files whose contents overlap with the 1.5 GB file. This simulates the typical Linux mirror setup with one DVD and many CD ISOs. To include aliasing effects, we duplicate the set and place one copy in a different directory, raising the total content size 6 GB.

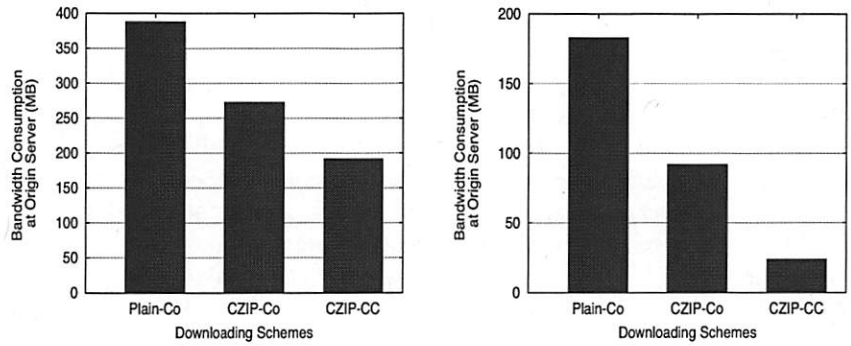
Our client workload is generated by six machines with one Gigabit Ethernet NIC each, split across two LAN switches. Each machine generates multiple simultaneous requests to the server, and we simulate 100-300 clients total. A new simulated client arrives on average every 3 seconds, up to the per-experiment client limit.

# of clients	Avg	Min	Median	90%
100	33.45	16.72	30.12	46.00
100 (w/CBN)	75.69	49.52	62.48	117.76
200	24.11	11.92	21.28	36.72
200 (w/CBN)	53.23	29.20	41.68	83.36
300	19.14	8.96	15.84	32.04
300 (w/CBN)	40.30	18.64	29.44	67.60

Table 3: Per-client throughputs (in Mbps) for serving a large data set with lots of commonality. We show the average, minimum, median, and 90th percentiles.

Figure 9 shows the client throughput CDF for serving 100, 200, and 300 simultaneous clients. The CZIP-enabled Apache outperforms the standard Apache by a factor of 2.11 to 2.26 (mean), and 1.84 to 2.07 (median). The means are higher than the medians because the CBN-cache case has a long tail – later requests do not overlap with many of the previous requests, so they compete less for the server's CPU cycles and network bandwidth. The CBN-cache case performs much better with physical memory cache hits than the non-cache case, whose throughput is bounded by the disk read bottleneck. This is observable in all three graphs by noticing that the horizontal gap between the two lines widens beyond the 50th percentile. Another interesting observation is that the worst-case throughput with the CBN cache beats 65% to 91% of non-cache throughputs.

The development effort for the components mentioned above were relatively modest. The Apache module consists of 700 lines of C, of which 235 are semicolon-containing. The CBN cache server is larger, with 613 semicolon lines out of 2061 total. However, the total development time for them was one week combined, and it was performed by a first-year graduate student. We believe that future development can leverage the effort here, especially of the stand-alone CBN cache.



(a) Downloading 100MB file

(b) Downloading 50MB file

Figure 10: Bandwidth Consumption at Origin Server: Plain-Co and CZIP-Co both use regular CoBlitz but Plain-Co downloads original file while CZIP-Co downloads CZIP'ed file. CZIP-CC uses C-CoBlitz to download the CZIP'ed file.

6.2 CBN-Aware Content Distribution

To build a CBN-Aware Content Distribution Network (CDN), we create C-CoBlitz by integrating CZIP support into the CoBlitz CDN, a scalable large-file HTTP CDN running on PlanetLab [15]. CoBlitz has been in production for over two years, and serves roughly 1 TB per day, with peaks as high as 5 TB/day and sustained bandwidth rates in excess of 1.2 Gbps.

CoBlitz already tries to reduce origin server bandwidth consumption, even when hundreds of edge proxies are serving multi-GB files. Rather than fetching whole files, CoBlitz nodes request and cache fixed-sized chunks of a file from the origin server using HTTP byte-range support. The nodes then cooperate with each other to reassemble the chunks in order and seamlessly serve the file to unmodified Web clients. Since CoBlitz does not examine the content of chunks, it does not identify redundant content. By integrating CZIP into CoBlitz, we can avoid fetching and storing redundant content, reducing origin server bandwidth consumption even further.

CBN cache integration with CoBlitz transparently provides the benefits described in Section 4.2 without any modification of the client or the server. CoBlitz names its fixed-size chunks using the original URL name and byte-range information. To add CZIP support, we add the content hash and chunk size to this name when handling CZIP-format files. The CBN cache integration requires only about 200 lines of new code.

The chunk naming scheme extends to the underlying CoDeeN content distribution network [31] on which CoBlitz is built. The benefit of this approach is that if a given chunk is not found at a CoDeeN node, the request is served from the peer CoDeeN node responsible for that given URL. The mapping of URLs to CoDeeN nodes is performed using the HRW consistent hashing scheme [27], so if a chunk is needed at several nodes, it will likely be fetched from a peer CoDeeN node instead of from the origin server.

To test how much bandwidth is saved using C-CoBlitz, we have 100+ U.S. PlanetLab nodes simultaneously download the first 100 MB and 50 MB of the Fedora Core 5 DVD ISO from a server at Princeton. For the origin server, we use lighttpd [9] on a 2.8 GHz Pentium D machine with 2 GB of memory. We download the 100 MB content first and then the 50MB content, for both the original and CZIP'ed versions. Using CZIP reduces the 100 MB and 50 MB files to 68.1 MB and 28.9 MB, respectively.

Figure 10 compares the bandwidth consumption at the origin server for the different schemes. Because the number of clients varies from 93 to 106 depending on the time of the tests, we normalize the bandwidth consumption for the 100-client case. Distributing the uncompressed 100 MB content to 100 nodes via regular CoBlitz consumes 388 MB of bandwidth at the origin server (3.8 copies of original content) while the CZIP'ed content needs 273 MB, a 29.6% reduction from serving the original content. This saving comes directly from the content size reduction by CZIP compression, and the same trend is seen in the case of the 50 MB file as well, which is reduced by 49.7%. Serving the CZIP'ed 50 MB content through C-CoBlitz shows the largest bandwidth reduction because most chunks were already cached while downloading the CZIP'ed 100 MB content. To serve the CZIP'ed file to 100 clients (2.9 GB of content or 5 GB uncompressed content), C-CoBlitz requires only 24 MB of origin server bandwidth, which is just 0.8% of the total size. Regular CoBlitz requires 3.83 times (92 MB) more bandwidth.

The other interesting comparison in Figure 10 is the bandwidth consumption drop from CZIP-CO (273MB) to CZIP-CC (191MB). The difference here is that CZIP-CO is just the CZIP'ed file being served over regular

CoBlitz, while CZIP-CC uses C-CoBlitz. Since this is the first transfer of the file to all 100 clients, these results should be the same, but C-CoBlitz still shows a 29.7% drop in bandwidth consumption. The C-CoBlitz system appears to be reducing the bandwidth burstiness and network congestion compared to regular CoBlitz, triggering fewer retries and causing each fetched chunk to be served to more peers. As a result, fewer nodes are fetching each chunk from the origin server, reducing the bandwidth consumption further.

7 Related Work

The idea of exploiting chunk-level commonality has been widely applied in many systems, but these techniques have not been easily separable from the underlying system. The earliest work of which we are aware is the system proposed by Spring *et al.* to eliminate the packet-level redundancy by recognizing identical portions in IP packets [26]. They assume synchronized caches at both endpoint routers and detect identical chunks by finding anchors [10] in the packet and expanding the region of same content from that point. The packet is encoded with tokens representing the repeated strings in the cache. Because the approach is independent of application-level protocols, it can offset application-level caching, such as removing the redundancy among the uncacheable HTTP responses. In CZIP, we focus on separating the CBN techniques from the underlying system, allowing us to provide similar benefits using only user-level CBN caches.

Another system that appeared shortly thereafter was LBFS [13], and was the first system taking advantage of Rabin's fingerprints to reduce bandwidth consumption in a distributed file system. LBFS finds about 20% redundancy in a 384 MB set of file data. Similar ideas have since been applied in numerous other file or storage back-up systems such as Farsite [1], Pastiche [5], Venti [19], CASPER [29], and Shark [2]. File systems are an attractive place to implement CBN caching because file access patterns often reveal significant redundancy [23, 30]. However, the CBN support has been built into these systems, making it more difficult to select when it is appropriate, or to use it outside of the range of tasks handled by the system.

Some systems have been built to reduce duplicate data specifically in the context of the Web. Value-based Web Caching (VBWC) [21] reduces redundant chunk transfer on the Web by coordinating the browser's CBN cache and a client-side parent proxy. Its operation is similar to Spring's work [26] except it is specific to HTTP. They also describe a synchronization mechanism between these two caches. Duplicate Transfer Detection (DTD) [12] adds a message digest in the HTTP response header (without a message body), and allows the client to search its CBN cache for the message body. Only in the case of a cache miss does the client ask for the message body – this is called the "pure-proceed" model. It is similar to our client-only caching scenario in Section 4.2, but the content hash is based on the whole file (except for byte-range queries), which may make finding redundancy across parts of files difficult. It does not require cache synchronization as in VBWC, but at the cost of an extra RTT delay for every cache miss. CZIP-based requests can utilize partial content overlap but do not require an extra RTT even for client-side caching only.

Delta encoding has also been proposed for Web pages that partially change [11]. In this general approach, clients can specify what version of a page they have cached when asking the server if the page has been updated. The server can then send just the updated portions rather than the whole page. While this approach is well suited for static content that has a small set of easily-identifiable versions, it is much harder to adapt it for dynamic content, which can not be easily named or tagged. The extra overhead on the server created a higher barrier to adoption for this approach. CZIP-based schemes would not have to remember specific states of Web pages, since the chunks can come from any page and be used in any other page. As a result, while a CZIP-based approach may not produce deltas as small as other approaches, it can find commonality across files with relatively little server state.

More recently, a transparent transfer service architecture based on a CBN chunk cache has been proposed [28]. It asks for a chunk hash array exchange before actual delivery, which is similar to the case in Section 4.3. Our belief is that by making CZIP a standard format, the benefits of this kind of compression can be achieved end-to-end, instead of just by an enhanced system in the middle. This approach would also let the endpoints select when to use it, avoiding the overhead when serving content with little data commonality, like unrelated compressed files [16].

Similarity-Enhanced Transfer (SET) [18] exploits chunk-level similarity in downloading related files. It finds relatively high chunk-level similarity in popular music and video files. Much of the similarity comes from files with the same content but with slightly different metadata information in the header. In order to utilize the similarity, SET proposes to maintain the CBN information and chunk location in a DHT-like infrastructure so that a SET-based downloader can easily find the chunk location with a constant number of lookups. CZIP-aware CoBlitz provides similar benefits without maintaining separate mapping information since the chunk data itself is cached with its metadata at the same location. This approach avoids any possible staleness concerns.

8 Conclusion

Although content-based naming (CBN) has proven itself useful in a variety of systems, there has been no general-purpose tool to enable its use in a variety of systems that were not designed with in mind from the start. In this paper, we have shown a new compression format, CZIP, which can be used to efficiently support CBN with reasonable overheads in processing power and space consumption. We have demonstrated that CZIP can identify and eliminate redundant data across a range of useful scenarios, without being tied to any particular system.

CZIP provides a flexible combination of deployment paths, not only providing benefits by itself, but also by providing more benefits if CZIP-awareness is added to the client, server, or both endpoints. We have described how these deployment options can be implemented without invasive changes to the endpoints. To support these claims, we have implemented CZIP awareness in the Apache Web Server, and have shown that integration with a CBN cache reduces its memory footprint and dramatically improves client throughput. We have also added CZIP support to a deployed content distribution network, and have shown that it reduces origin server bandwidth consumption significantly.

We believe that this combination of flexibility, ease of integration, and performance/consumption benefits will make CZIP an attractive tool for those wishing to support content-based naming or develop new systems using this technique.

Acknowledgments

We thank Jeff Sedayao and Claris Castillo for providing the realistic VM images. We also want to thank Sanjay Rungta and Michael Kozuch for useful discussion about redundancy suppression in enterprise network traffic as well as in virtual machine migration. This work is supported in part by NSF grants CNS-0615237 and CNS-0519829.

References

[1] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J.Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[2] S. Annapureddy, M. J. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, May 2005.

[3] H. Bahn, H. Lee, S. Noh, S. Min, and K. Koh. Replica-aware caching for web proxies. *Computer Communications*, 25(3):183–188, 2002.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'03)*, 2003.

[5] L. Cox and B. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[6] Data Domain. `http://www.datadomain.com/`.

[7] J. Douceur, A. Adya, W. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[8] EMC Corporation. `http://www.emc.com/`.

[9] J. Kneschke. lighttpd. `http://www.lighttpd.net`.

[10] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, 1994.

[11] J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein. Delta encoding in HTTP. RFC 3229, January 2002.

[12] J. C. Mogul, Y. M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, 2004.

[13] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'01)*, pages 174–187, 2001.

[14] P. Nath, M. A. Kozuch, D. R. O'Hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proceedings of USENIX Annual Technical Conference*, 2006.

[15] K. Park and V. S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of the 3rd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '06)*, 2006.

[16] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of USENIX Annual Technical Conference (USENIX '04)*, 2004.

[17] F. Project. `http://fedora.redhat.com/`.

[18] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proceedings of the 4th USENIX/ACM Symposium on Networked Systems Design and Implementation(NSDI'07)*, 2007.

[19] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of First USENIX conference on File and Storage Technologies*, 2002.

[20] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard University, 1981.

[21] S. Rhea, K. Liang, and E. Brewer. Value-based web caching. In *Proceedings of the Twelfth International World Wide Web Conference*, May 2003.

[22] Riverbed Technology, Inc. `http://www.riverbed.com/`.

[23] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of USENIX Annual Technical Conference*, 2000.

[24] A. Rousskov, M. Weaver, and D. Wessels. The fourth cache-off. `http://www.measurement-factory.com/results/`.

[25] M. Satyanarayanan, M. Kozuch, C. J. Helfrich, and D. R. O'Hallaron. Towards seamless mobility on pervasive hardware. *Pervasive and Mobile Computing*, 1(2):157–189, July 2005.

[26] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, 2000.

[27] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, Feb. 1998.

[28] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for internet data transfer. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, 2006.

[29] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, and A. Perrig. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of USENIX Annual Technical Conference(USENIX '03)*, 2003.

[30] W. Vogels. File system usage in windows NT 4.0. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 1999.

[31] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[32] Wikipedia. `http://www.wikipedia.org/`.

Implementation and Performance Evaluation of Fuzzy File Block Matching

Bo Han
Department of Computer Science
University of Maryland
College Park, MD 20742, USA
bohan@cs.umd.edu

Pete Keleher
Department of Computer Science
University of Maryland
College Park, MD 20742, USA
keleher@cs.umd.edu

Abstract

The fuzzy file block matching technique (fuzzy matching for short), was first proposed for opportunistic use of Content Addressable Storage. Fuzzy matching aims to increase the hit ratio in the content-addressable storage providers, and thus can improve the performance of underlying distributed file storage systems by potentially saving significant network bandwidth and reducing file transmission costs. Fuzzy matching employs *shingling* to represent the fuzzy hashing of file blocks for similarity detection, and error-correcting information to reconstruct the canonical content of a file block from some similar blocks. In this paper, we present the implementation details of fuzzy matching and a very basic evaluation of its performance. In particular, we show that fuzzy matching can recover new versions of GNU Emacs source from older versions.

1 Introduction

Recent work in file systems has shown that the use of *content-addressable storage* (CAS) can enhance the performance of distributed file systems, especially in the wide area [7, 11, 16]. The basic idea of CAS is to describe files in terms of *recipes* that enumerate a set of blocks that make up the file's contents. Similar files, or different versions of the same files, may contain blocks in common. CAS-based file systems can exploit this property by requesting remote copies of only those blocks that are not already present locally, or at least nearby.

Tolia et al. [16] proposed extending this technique by using *fuzzy matching* to identify local blocks similar to a target block, and then using error correcting codes to *correct* such a local block to the target. However, Tolia did not present any experimental data on how well the idea works in practice, nor any implementation details.

The purposes of this brief paper are to (1) provide an existence proof of the idea, (2) describe our approach in enough detail to enable other researchers to extend it, and (3) make a preliminary evaluation of its performance on real data. We make no claim here of exhaustive analysis or experiments. However, our results do show that the technique has promise, and can be highly beneficial in the right circumstances. In particular, we evaluate the use of fuzzy matching to reconstruct later versions of GNU Emacs source [1] from earlier versions. We also evaluate the performance impact of varying several important parameters, such as average block size, the number of subblocks per block and error correcting code rate, etc.

The rest of this paper is organized as follows. In Section 2, we introduce the background of fuzzy matching. We next describe details of our implementation in Section 3. In Section 4 we evaluate the performance of fuzzy matching and the effect of several parameters. After summarizing related work in Section 5, we conclude and present our future directions in Section 6.

2 Fuzzy Matching

In this section, we describe fuzzy matching and subsidiary techniques in more detail. These techniques include Rabin fingerprints [14], shingling [3], and error correcting codes.

A file recipe in a standard CAS system consists of an ordered list of block signatures. The signatures are generated from the blocks' contents through a cryptographically secure hash, such as SHA-1. Without fuzzy matching, this hash value is sufficient to completely identify each block needed to reconstruct the file.

Fuzzy matching extends the description of each block into a specification that includes four pieces of information: (a) an exact hash value that matches only the correct block; (b) a fuzzy hash value that matches blocks similar to the correct block; (c) fingerprints of a block's fixed-length subblocks for identifying them in similar blocks, and (d) error-correcting information that may

sometimes recover the correct block, when applied to a similar block.

Fuzzy matching works as follows: (1) The client sends the target block's recipe to its nearby CAS provider (which may be local). (2) The CAS provider first determines whether it holds a file block whose hash matches the exact hash value of the requested block; if so, it returns this block to the client. (3) If the CAS provider doesn't hold the correct block, it next uses the block's fuzzy hash value to identify any candidate blocks that approximately match the file block requested by the client. (4) The CAS provider applies the error-correcting information to each such candidate block. If the corrected block's hash matches the exact hash value of the requested block, the CAS provider returns the corrected block to the client. (5) If none of the CAS provider's candidate blocks can be corrected to match the exact hash, the CAS provider returns a negative result to the client, which then sends a request to a remote file server [16].

2.1 Rabin Fingerprints and Shingling

Fuzzy matching uses Rabin fingerprints to construct content-defined data blocks, and to compute shingles for similarity detection. Fingerprints are short tags for large objects. The property of fingerprints is that if two fingerprints are different then the corresponding objects are certainly different. The probability that two different objects have the same fingerprint is extremely small. For more information about Rabin fingerprints, please refer to [14].

Shingling was proposed by Broder et al. to determine the syntactic similarity of web pages [3]. They view each web page as a sequence of words, and a contiguous subsequence contained in the web page is called a *shingle*. Fingerprints of shingles are computed using sliding window to efficiently create a shingling vector for a web page. Instead of comparing entire documents, they use shingling vectors to measure the resemblance and containment of documents in a large collection of web pages. Shingling is also called super-fingerprint in REBL [8]. Fuzzy matching selects the s smallest fingerprints among the shingling vector to form a *shingleprint*, and stores it in the block recipe as the fuzzy hash value of a block.

2.2 Error Correcting Codes

The essence of fuzzy matching is to store enough error-correcting information to recover the original data from similar blocks. Therefore, we give a brief introduction of Error Correcting Code (ECC) in this subsection. An error correcting code is a code which constructs data signals conforming to specific rules, such that errors in the received signal can be automatically detected and corrected. There are two important subclasses of error correction: Forward Error Correction (FEC) and Backward Error Correction (BEC). In the following, we will focus on FEC which is suitable for fuzzy matching. FEC is accomplished by adding redundancy to data bits using a predetermined algorithm. The two main categories of FEC are block coding and convolutional coding.

A (n, k) block code contains sequences of n symbols. Each sequence of length n is a code word or code block, and contains k information digits. The remaining $n - k$ digits are called redundant digits. Here, the code rate is defined as the ratio $R = k/n$. Examples of block codes include (7, 4) and (11, 7) Hamming code which can correct single-bit errors and detect double-bit errors; (23, 12) and (11, 6) Golay code which can correct 3 and 2 errors, respectively; (255, 223) and (65535, 65503) Reed-Solomon code which can correct 16 errors and 32 erasures (errors whose locations are known in advance). Fore more detailed information about Error-Correction Coding, we refer the interested reader to [4, 10]. In our implementation, we use a (255, 223) Reed-Solomon code for its higher code rate and error-correction capability. Evaluating the performance of other error correcting codes is part of our future work.

3 Implementation Details

Our implementation of fuzzy matching has two main building blocks: recipe creation and file block reconstruction. In the following, we describe the details of these two parts, respectively.

3.1 Constructing File Recipes

The first step of constructing file recipes is to divide a file into variable-length content-defined blocks using Rabin fingerprints. Content-defined chunking (CDC) has been used in LBFS [11], Pastiche [5] and TAPER [7]. CDC sets block boundaries based on file contents, rather than on position within a file. Therefore, insertions and deletions can only affect the surrounding blocks and not the entire file. A sliding window is used to evaluate a fingerprint of the preceding w bytes at each point in the file. A point is considered to be a boundary of a data block if its 64-bit Rabin fingerprint matches a predetermined marker value. Rabin fingerprints are chosen because they are efficient to compute on a sliding window over a file. The number of bits in a Rabin fingerprint that are used to match the marker determines the expected block size. For example, if the low-order l bits are used, the expected block size will be 2^l.

After a file is divided into variable-length blocks, a shingleprint is computed for each block. Define a shingle to be a sequence of m contiguous bytes in a block.

There are totally $B-m+1$ overlapping shingles in a B-byte block. We again use Rabin fingerprints to compute a hash of each shingle in the block. These fingerprints are then sampled to compute a shingleprint of the block. We employ Min_s sampling [3] which selects the set of s fingerprints with the smallest values to represent the approximate content of a block. Two blocks are similar if they share in common t (similarity threshold) out of s values in the shingleprint. Note that bloom filters could also be utilized for similarity detection [7]. In the future, we plan to compare these two approaches associated with fuzzy matching to understand the relative detection performance.

The last step is to generate error-correcting information for each block. As mentioned above, we choose the (255, 223) Reed-Solomon code for its high code rate. Here, suppose a file block is equally divided into seven subblocks (because the ratio between the number of information digits and that of erasures which can be corrected is about 7). To identify these subblocks, the Rabin fingerprint of each subblock is first stored in the block recipe. Then another subblock is constructed to keep the error-correcting information. To do so, each subblock is further divided into several 31-byte pieces. The first seven pieces of each subblock are packed together to form a 217-byte data chunk. After padding with 6 null bytes, we get a 223-byte chunk and use the (255, 223) Reed-Solomon Code to compute the 32-byte error-correcting information. Finally this 32-byte data piece is put at its corresponding position in the ECC subblock. The same method is used to process other data pieces in each subblock to construct the ECC subblock for the entire data block. This procedure is also demonstrated in Figure 1.

3.2 Recovering from Similar File Blocks

Assume that a CAS provider has identified a candidate block by finding a shingleprint match of the target and candidate blocks. We use the example in Figure 1 to illustrate how the provider can then recover the target block from the candidate. Compared with the target block, the candidate block contains an deletion in the 3rd subblock. Suppose there are 1024 bytes in each subblock. The CAS provider computes the Rabin fingerprint for each contiguous 1024-byte subblock of the candidate block, and identifies those subblocks whose Rabin fingerprints match those provided by the client. In this case, the CAS provider can find out 6 unchanged subblocks: 1, 2, 4, 5, 6 and 7. These subblock correspondences between the two blocks are shown by vertical arrows in Figure 1. Thereafter, using the error-correcting subblock, the CAS provider can overcome the "erasure" (the missing 3rd subblock) of the modified subblock, to form the target block. As a final check, the CAS provider computes the exact hash over the entire corrected block, and compares it with the exact hash of the target block.

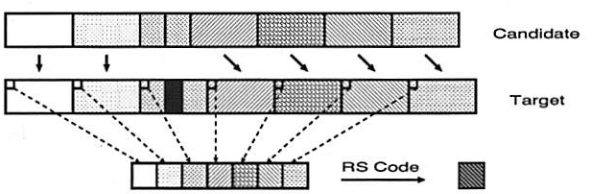


Figure 1: Generating ECC subblock and recovering a block from a similar block

3.3 ECC Subblock Formation Schemes

As mentioned above, each content-defined block is further divided into seven subblocks for generating the ECC subblocks. This approach can only correct from candidate blocks with modifications to a single subblock. Increasing the number of subblocks can potentially allow more changes to be corrected by separating their error correction information. Therefore, we also propose two other approaches, *Separate Selection* and *Grouped Selection*, to dividing each block into $b = 7i$ (for some small non-negative integer i) subblocks. For example, when $b = 14$, *Separate selection* generates a single ECC subblock for subblocks 0, 2, 4, 6, 8, 10 and 12, and another one for subblocks 1, 3, 5, 7, 9, 11 and 13. *Grouped selection* creates a ECC subblock for the first seven subblocks, and another one for the last seven subblocks.

4 Performance Evaluation

This section provides a performance evaluation of fuzzy matching using five different releases of Emacs source: emacs-21.4, emacs-21.3, emacs-21.2, emacs-21.1 and emacs-20.7. We study the effect of several parameters on the performance of fuzzy matching, investigating primarily along two dimensions: the size of file recipes (overhead) and the probability that a file in the new release can be recovered from those in the adjacent old version (effectiveness).

4.1 Parameter Study

Fuzzy matching's performance depends on several parameters: the sliding window size w, the average block size 2^l, the sliding window size for shingling m, the number of shingles in a shingleprint s, similarity threshold t, and the number of subblocks per block b. Our default values are: $w = 48$, $2^l = 4,096$, $s = 10$, $t = 8$, $m = 12$ and $b = 7$.

	Sliding Window Sizes			
	12B	24B	48B	96B
21.3→21.4	9	9	9	10
21.2→21.3	243	243	246	237
21.1→21.2	185	179	175	182
20.7→21.1	142	134	143	133

Table 1: Number of recovered files for various sliding window sizes for CDC.

4.1.1 Sliding Window Size for Content-Defined Chunking

The size of the sliding window can determine how effective the chunking algorithm is in defining block boundaries similarly, despite intervening edits. Table 1 shows a sampling of our results for the number of recovered files versus different window sizes. The recovered files are files that can be corrected from their similar (not exactly identical) old versions. The reason for the small numbers in the first row is that there are only 10 different files between emacs-21.3 and emacs-21.4. The results change little, showing that the chunking mechanism is relatively insensitive to the window size.

4.1.2 Average Block Size

Table 2 summarizes the number of recovered files for different average block sizes. Larger blocks can increase the size of subblock which will potentially make changes occur in one subblock rather than span several subblocks. Therefore, they can improve the probability that a block can be recovered from a similar block. Moreover, larger blocks require less overhead to track the exact hash value and numerous shingleprints and fingerprints per file which is also verified by our experiment results (not shown here for space limitation). In addition, larger blocks also decrease the number of comparisons. The possible reason for the exception in the last row of Table 2 is that changing average block size may also alter the block and subblock boundaries which will sometimes reduce the file recovery probability.

	Average Block Sizes					
	0.5K	1K	2K	4K	8K	16K
21.3→21.4	9	9	9	9	9	9
21.2→21.3	224	231	251	246	246	245
21.1→21.2	152	154	163	175	186	191
20.7→21.1	151	151	148	143	144	140

Table 2: Number of files that can be recovered from similar files for various average block sizes.

	Similarity Thresholds					
	0	2	4	6	8	10
21.3→21.4	9	9	9	9	9	6
21.2→21.3	248	247	246	246	246	205
21.1→21.2	180	179	178	178	175	132
20.7→21.1	147	147	147	147	143	100

Table 3: Number of recovered files for different similarity thresholds.

4.1.3 Sliding Window Size for Shingling

This sliding window size is, in fact, the shingle size. A shingle should be large enough to create many possible substrings, which minimizes spurious matches, and small enough to prevent small modifications from affecting many shingles. Common values in past studies have ranged from four to twenty bytes [8]. Our experimental results (omitted for space) indicate that shingling window size does not significantly affect the performance of fuzzy matching.

4.1.4 Similarity Threshold

Shingling is used to identify candidate blocks. The similarity threshold is the number of fingerprints in a shingleprint that must match to declare two blocks similar. Table 3 reports the number of recovered files for different similarity thresholds. The last three rows illustrate that reducing similarity thresholds can slightly increase the number of recovered files. The reason is that lower thresholds lead to the identification of more candidate blocks. However, larger candidate sets lead to more computational overhead. Given that the number of recovered files is nearly identical for all but the last column, setting the threshold near, but not equal to the number of shingles in the shingleprint, seems to be a good compromise.

4.1.5 Number of Subblocks per Block

Figure 2 and Figure 3 present the number of recovered files with differing numbers of subblocks, for *grouped selection* and *separate selection*, respectively. Using small subblocks allows the algorithms to tolerate more errors (modifications) because there is another ECC subblock for each seven subblocks in the block. One drawback is that a single error, which may have been contained in a single large subblock, can span several small subblocks. With grouped selection, neighboring subblocks are usually corrected by a single ECC subblock. Recall that ECC subblocks can only correct a single faulty subblock, so errors spanning consecutive subblocks usually result in a correction failure for grouped selection. Separate selection does a better job in this case

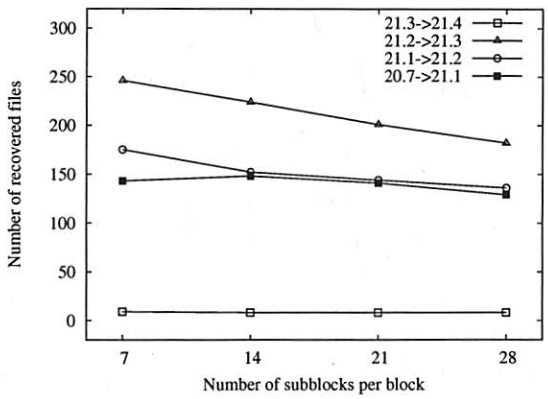


Figure 2: Number of recovered files for different number of subblocks using grouped selection

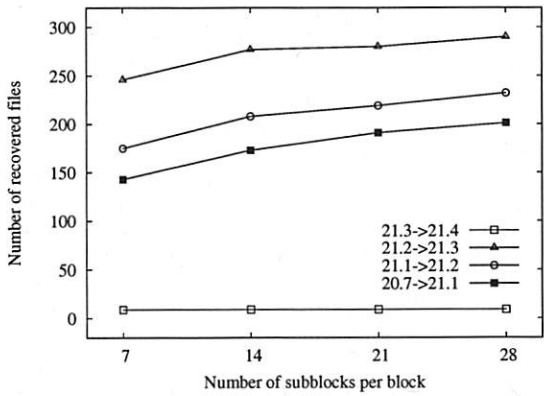


Figure 3: Number of recovered files for different number of subblocks using separate selection

because neighboring subblocks are always protected by different ECC subblocks. In either case, recipe size increases linearly with the number of subblocks.

4.2 Recovering One File from All Files

The above experiment only attempts to recover a file from older versions with the same path names. For example, file `emacs-21.4/src/coding.c` is compared with `emacs-21.3/src/coding.c`. We would like to answer the following, more general, question: given a target file F and a corpus of files S, with what probability can we find another file F' in S which is similar enough to F? That is, given the error-correcting information of F, we can reconstruct F from F'. We randomly selected 100 files from emacs-21.3 and attempted to recover each from the set of all files in emacs-21.2, excepting the file with the same path name. Using separate selection with 28 subblocks, we were only able to recover a single file in its entirety.

	R #	R	S %	B #	E #
21.3→21.4	9	18.00	99.81	10	1
21.2→21.3	296	17.92	58.08	2345	1678
21.1→21.2	243	17.98	62.11	2187	1525
20.7→21.1	213	13.85	7.39	4002	3523

Table 4: Summary of more detailed experiment results.

4.3 Similarity of Blocks in Different Files

The above experiment was performed at the level of complete files. We carried out a similar experiment at the block level, attempting to recover 100 random blocks from emacs-21.3 from the set of blocks contained in emacs-21.2, again excepting only blocks from the same file. We were only able to recover two such blocks.

4.4 More Detailed Experiment Results

Table. 4 shows more detailed results for separate selection with the following parameters: $w = 48$, $2^l = 16,384$, $t = 8$, $m = 12$ and $b = 28$. 'R #' is the number of files successfully recovered. The second column shows that the average sizes of the fuzzy file recipes are less than or equal to 18% of the full file sizes in all cases. 'S %' is the size percentage of recovered files plus unchanged files, and hence is an indication of potential savings. The fourth column shows the total number of blocks in the unrecoverable files, and the fifth column shows the number of unrecoverable blocks. For example, a single unrecoverable block is preventing a 10-block file from being recovered in the emacs-21.4 distribution.

5 Related Work

Fuzzy matching utilizes SHA-1 hash of a block's contents as the exact hash value. The use of content-hash to uniquely identify blocks (compare-by-hash) has been widely explored previously. For example, a unique hash of a block's contents is used as the block identifier for read and write operations in Venti [13]. Pastiche also employs content-hash to find redundant data across versions of files for backup [5].

Finding similar data blocks has also been extensively studied. Policroniades and Pratt evaluate three techniques for discovering identical pieces of data: whole file content hashing, fixed size blocking, and a chunking strategy [12]. Kulkarni et al. propose a scheme, called Redundancy Elimination at the Block Level (REBL), for storage reduction [8]. REBL uses super-fingerprints to reduce the data needed to identify similar blocks. TAPER [7] provides a content-based similarity detection technique which uses Bloom filters [2] to identify similar files.

In conjunction with conventional compression and caching, the Low Bandwidth File System (LBFS) [11] takes advantage of commonality between distinct files and successive versions of the same file in the context of a distributed file system. Lee et al. describe a technique, called operation-based update propagation, for efficiently transmitting updates to large files that have been modified on a weakly connected client of a distributed file system [9]. They also use error correcting codes to correct short replacements in similar blocks.

6 Conclusions

In this paper we describe the design, implementation, and performance of a fuzzy file block matching scheme. The main advantage of fuzzy file recipes is in saving network bandwidth as, for purposes of a wide-area file system, we can treat CPU cycles and disk space as effectively being free. If we accept the percentages shown in Table 4, the average file recipe size is about 18% of the size of the corresponding file. Hence, approximately one in five of recipes transmitted across the network must be "useful" (prevent a subsequent download of the corresponding file) in order for the system overall to reduce network bandwidth.

Our results anecdotally show that fuzzy file recipes are seldom able to find matches among random blocks. Instead, the utility of this approach would seem to lie in finding and exploiting commonality among different versions of the same files. For example, the distributer of a new version of the GNU Emacs source might preprocess files, identifying those files that can be recreated from one or more earlier versions of the source. Only those files would be included as fuzzy file recipes; the others would be distributed as either patches or complete copies. Though explicit patches would generally take less space than fuzzy file recipes, patches are only useful if the recipient has the exact version referenced by the patch. In the absence of complete information, fuzzy file recipes would be preferable.

Fuzzy file recipes would also be useful for versioning file systems [15]. Such systems are becoming more common as increasing disk capacities remove the incentive to destroy old file versions. In future work, we plan to expand our data set, try other error correcting codes, and experiment with more parameter combinations. Finally, we plan to integrate fuzzy matching into an existing distributed file system, such as MoteFS [6].

7 Acknowledgments

We thank Niraj Tolia and Kan-Leung Cheng for their valuable feedback and suggestions. We thank Henry Minsky for making the implementation of Reed-Solomon Code available and Hyang-Ah Kim for making the Rabin fingerprints code available. We also thank Benjie Chen et al. for opening the source code of their Low-Bandwidth Network File System.

References

[1] Gnu emacs, http://www.gnu.org/software/emacs/.

[2] BRODER, A., AND MITZENMACHER, M. Network Applications of Bloom Filters: A Survey. *Internet Mathematics 1*, 4 (2005), 485–509.

[3] BRODER, A. Z., GLASSMAN, S. C., MANASSE, M. S., AND ZWEIG, G. Syntactic Clustering of the Web. In *Proceedings of the 6th International WWW Conference* (April 1997).

[4] CLARK, G. C., AND CAIN, J. B. *Error-Correction Coding for Digital Communications*. New York: Plenum Press, June 1981.

[5] COX, L. P., MURRAY, C. D., AND NOBLE, B. D. Pastiche: Making Backup Cheap and Easy. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 285–298.

[6] GABURICI, V., KELEHER, P., AND BHATTACHARJEE, B. File System Support for Collaboration in the Wide Area. In *Proceedings of the 26th International Conference on Distributed Computing Systems* (Lisboa, Portugal, July 2006).

[7] JAIN, N., DAHLIN, M., AND TEWARI, R. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies* (December 2005), pp. 281–294.

[8] KULKARNI, P., DOUGLIS, F., LAVOIE, J., AND TRACEY, J. M. Redundancy Elimination within Large Collections of Files. In *Proceedings of the USENIX 2004 Annual Technical Conference* (Boston, MA, June 2004), pp. 59–72.

[9] LEE, Y.-W., LEUNG, K.-S., AND SATYANARAYANAN, M. Operation-based Update Propagation in a Mobile File System. In *Proceedings of the USENIX 1996 Annual Technical Conference* (Monterey, CA, June 1996), pp. 43–56.

[10] MOON, T. K. *Error Correction Coding: Mathematical Methods and Algorithms*. Wiley-Interscience, June 2005.

[11] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Chateau Lake Louise, Banff, Canada, October 2001), pp. 174–187.

[12] POLICRONIADES, C., AND PRATT, I. Alternatives for Detecting Redundancy in Storage Systems Data. In *Proceedings of the USENIX 2004 Annual Technical Conference* (Boston, MA, June 2004), pp. 73–86.

[13] QUINLAN, S., AND DORWARD, S. Venti: A New Approach to Archival Storage. In *Proceedings of the First USENIX conference on File and Storage Technologies* (January 2002), pp. 89–101.

[14] RABIN, M. O. Fingerprinting by Random Polynomials. Tech. Rep. TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[15] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding When to Forget in the Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (near Charleston, SC, December 1999), pp. 110–123.

[16] TOLIA, N., KOZUCH, M., SATYANARAYANAN, M., KARP, B., PERRIG, A., AND BRESSOUD, T. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *Proceedings of the USENIX 2003 Annual Technical Conference* (San Antonio, TX, June 2003), pp. 127–140.

From Trusted to Secure: Building and Executing Applications that Enforce System Security

Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel
Systems and Internet Infrastructure Security Laboratory (SIIS)
Computer Science and Engineering, Pennsylvania State University
{*phicks,ruedarod,mcdaniel,tjaeger*}*@cse.psu.edu*

Abstract

Commercial operating systems have recently introduced mandatory access controls (MAC) that can be used to ensure system-wide data confidentiality and integrity. These protections rely on restricting the flow of information between processes based on *security levels*. The problem is, there are many applications that defy simple classification by security level, some of them essential for system operation. Surprisingly, the common practice among these operating systems is simply to mark these applications as "trusted", and thus allow them to bypass label protections. This compromise is not a limitation of MAC or the operating system services that enforce it, but simply a fundamental inability of any operating system to reason about how applications treat sensitive data internally—and thus the OS must either restrict the data that they receive or trust them to handle it correctly.

These practices were developed prior to the advent security-typed languages. These languages provide a means of reasoning about how the OS's sensitive data is handled *within* applications. Thus, applications can be shown to enforce system security by guaranteeing, in advance of execution, that they will adhere to the OS's MAC policy. In this paper, we provide an architecture for an operating system service, that integrate security-typed language with operating system MAC services. We have built an implementation of this service, called SIESTA, which handles applications developed in the security-typed language, Jif, running on the SELinux operating system. We also provide some sample applications to demonstrate the security, flexibility and efficiency of our approach.

1 Introduction

The problem of building secure systems with mandatory policy to ensure data confidentiality and integrity is coming into the forefront of systems development and research. Mandatory access controls (MAC) for type enforcement (TE) along with support for multi-level security (MLS) are now available in the mainline Linux distributions known as Security Enhanced (SE)Linux [25]. Trusted Solaris [21] and TrustedBSD [9] also provide MAC security. A recent release [35] made SELinux-like security available for Mac OS X, as well. Other projects such as NetTop [19] (which is being built with SELinux) seek to provide strong assurance of data separation. The goals of data separation (called *noninterference* elsewhere in the literature [10]) hold an important place also in the recent efforts towards virtualization with VMware [8], Xen [2] and others. All of these efforts run into a critical problem—they seek to enforce security only at the granularity of application inputs and outputs. *They cannot monitor how data is handled within an application.*

This approach would be acceptable if each application instance only handled data at a single security level. If that were the case, the operating system could prevent an application from reading or writing at any other level. The reality, however, is that many applications must simultaneously handle inputs and outputs with different security levels. This problem has led to two ad hoc approaches, both of which have serious limitations. The first approach sacrifices security to improve flexibility and efficiency. By marking an application as "trusted," it is given a special status to handle inputs and outputs with varying security policies. The operating system must then presume that the application will internally handle the data correctly. At best, the application's code is subjected to a manual inspection. The second solution compromises flexibility and efficiency in order to ensure security. Applications that must handle inputs with differing security levels are split into multiple executions, with one to handle each level. This complicates legitimate communication between processes and expends system resources, making it slow, error-prone and not always sufficiently expressive.

The first solution has been applied primarily to system utilities. A quick check indicates that SELinux, for

example, trusts dozens of applications (30-40 SELinux application types have special privileges for this purpose) to correctly handle data of multiple levels: consider `passwd`, `iptables`, `sshd`, `auditd`, and `logrotate`, to name a few (a complete list is given in Figure 1). As an example, `logrotate` handles the data from many different levels of logs as well as its own configuration files. It also runs scripts and can send out logs via email. Even after a thorough inspection of the code, it is hard to say with certainty that it never leaks log data to its (publically readable) configuration files or improperly sends mail to a public recipient (in fact, as of v. 2.7.1, this could actually happen).

There are a number of user applications, for which the second approach is more common. Email clients are a classical example, but web browsers, chat clients and others also handle secrets such as credit card numbers and passwords along with other mundane data. Some servers (web servers, chat servers, email servers) also fall into this category, handling requests from various levels of users and thus requiring multiple versions of the same application to run simultaneously.

What we would like is to be able to communicate the operating system's data labels (the label on files, sockets, user input, etc.) into the application and ensure that, throughout the application, the labeled data flows properly (i.e., in compliance with the operating system policy). Fortunately, a new technology has become available to aid in this process. Emerging security-typed languages, such as Jif [22], provide *automatic verification* of information flow enforcement *within* an application. Through an efficient, compile-time type analysis, a security-typed language compiler can guarantee that labeled data flowing through an application never flows contrary to its label. This provides a formal basis for trusting applications to handle data with multiple levels of sensitivity. Admittedly, these technologies are still in development and thus still challenging to use. Programming in Jif can be a frustrating endeavor. To aid this, we are investigating tools for semi-automatic labeling of programs. That said, we found that Jif is tractable in its current form for programming some small utilities and user applications which require handling of multi-level data. Such utility presents an as yet unrealized opportunity to improve broader systems security. To our knowledge, there has been no investigation of the ways in which these application guarantees could be used to augment greater system security.

To this end, we have designed and built an infrastructure that 1) allows an operating system with mandatory access controls to pass labeled data into an application and 2) to be certain that the data will not flow through the application contrary to the operating systems policy. For our investigation, we have focused on the most mature security-typed language, Jif/Pol (Jif enhanced with our policy system [13]) and the widely-studied, open source, SELinux operating system. Because these languages have not yet been widely used, there is no infrastructure available for interacting with secure operating systems. To remedy this, we have provided 1) an API for Jif by which labeled data such as sockets, files and user I/O can be received from and passed out of the OS—this API ensures consistency between operating system and application labels. 2) We provide a compliance analysis that ensures that the labeled data will be handled securely within the application, in compliance with the OS's mandatory policy. We integrated these changes into an operating system service we call SIESTA, that can be used to securely execute multi-level applications written in Jif, by first verifying that they will not violate the operating system's security policy.

To demonstrate the effectiveness of our approach, we used Jif/Pol to build some prototype applications: a security-typed version of the `logrotate` utility and an email client that can handle multiple email accounts of varying security levels. For `logrotate`, we were able to determine that it is possible to have total separation between log files of different programs and between log files and configuration files, so long as the configuration files have a lesser or equal confidentiality than the log files.

In this work, we make the following contributions. We identify a fundamental limitation in MAC systems security and we show how recent advances in programming languages can be applied to solve this problem. More specifically, we give a clean, general architecture for using security-typed language technology to enforce system security *within* applications. To test our approach, we implemented this architecture for Jif and SELinux and provide some reusable software artifacts. Namely, we extend the Jif Runtime environment to provide a reusable API for reading and writing OS resources labeled with SELinux security contexts. We also give a policy analysis which tests Jif policy for compliance with SELinux policy. Additionally, we provide a system service, SIESTA, that incorporates this analysis tool and uses it to determine whether a Jif application can be securely executed in a given SELinux operating system. Finally, we evaluate our implementation for security, flexibility and efficiency using some example applications we constructed.

In Section 2, we give some background on MAC security and security-typed languages, we also describe the problems involved with integrating security-typed languages into MAC OS's. We give our architecture in Section 3 and describe the implementation of this architecture and some demonstrative applications in Section 4. We evaluate these applications, as well as our approach,

for its usability, efficiency and security in Section 5. We examine some related work in Section 6 and we conclude in Section 7.

2 Problem

2.1 Security background

Security lattices Standard information flow models, on which we base our work, arrange labels on data as a lattice of principals, sometimes called a *principal hierarchy*. The traditional model [3, 7] allows data only to flow up the lattice (i.e. data can become more secure, but not less secure). If, for some reason, data must flow down the lattice, a *declassification* must take place. These policy violations should be infrequent or non-existent and if occurring at all should be carefully regulated. Filters for regulating declassification are called *declassifiers*.

Lattices may have a variety of principals and structures. A standard military lattice is simply a vertical line containing five levels[1]: unclassified, classified, confidential, secret and top secret, with top secret placed on the top of the lattice and unclassified on the bottom. While unclassified data can be written to classified files, the opposite is not true.

We use the term "MLS" broadly throughout this paper. Although the term has traditionally referred to military levels of sensitivity, such as secret or top secret, more general lattices can also be expressed [7]. For example, consider the lattice in Figure 3. In this lattice, data labeled `configP` can be written up to `xserver_log_t:s1`. The principals at the top of this lattice are all *incomparable* with each other. This means that data cannot flow between these labels at all. They could only be written to a mutually higher principal.

2.2 Enforcing MAC policies within applications

In an OS with MAC security, the OS can monitor its resources (such as files, sockets, etc.) and when an application tries to read them, write them, delete them, etc. it can prevent the application from performing one of these security-sensitive operations. To accomplish this, OS entities are divided into subjects and objects. Every operating system resource (socket, file, program, etc.) is an object, labeled with a type and an MLS level[2], such as `system_u:object_r:user_t:s0` for a public object owned by the reader. We will abbreviate this as `user_t:s0` since the user and role labels are always the same for system resources. The running process is considered a subject and also has a label, such as `system_u:system_r:logrotate_t:s0-s1`, where the colon-separated label consists of user, role, type and MLS level (reading the label from left to right). Notice that the MLS level may consist of a range, indicating that a particular process can handle a range of levels. In this case, the subject would have access to objects in the range `s0` to `s1`.

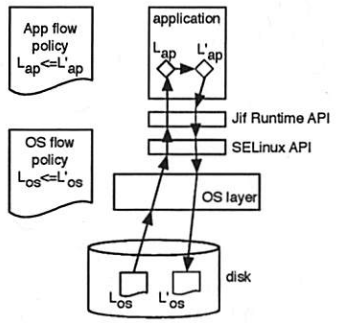


Figure 2: As data passes from a disk through the OS into an application and again when it is written back out, there must be consistency in labels and permitted flows at the OS and application levels. This requires proper labeling and compliance of the application policy with the operating system policy.

If a running process for `logrotate`, for example, has this label and attempts to read from a user's file labeled with `user_t:s4`, a security check will be triggered by this security-sensitive operation and (under a typical policy) it will be stopped by the Linux Security Module (LSM). However, if `logrotate` has permission to read from a log file labeled `var_log_t:s1` and to write to a configuration file labeled `logrotate_var_lib_t:s0` (which it normally does), the OS cannot stop it from reading the log data and leaking it down to the lower security configuration file. This could leak secrets stored in the log data to the publicly readable configuration file. Currently, the `logrotate` utility and the other utilities in Figure 1 are merely trusted not to leak data and it is not easy to verify, by manual inspection, that the C code for these utilities does not contain such a leak.

What is needed is 1) a way to pass the security labels into the application along with the resources, and 2) an automated way to ensure that the application honors the flow requirements on the labels. Furthermore, both of these conditions should be checked prior to executing the application. This situation is illustrated in Figure 2. Because the second requirement is precisely what security-typed languages, such as Jif, do well, we consider how they might be used to meet this need.

In Jif, when a variable is declared, it is tagged with a security label. An automated type analysis ensures no leakage can occur through implicit or explicit flows. For example, consider a program which has been executed by Alice (who can enter information through `stdin` and read from `stdout`), but which also has access to files, some of which can not be accessed by Alice (like persistent state such as statistics from others' executions) and some of which are publicly readable (like `config`

Type of utility	Trusted applications
Policy management tools	`secadm`, `load_policy`, `setrans`, `setfiles`, `semanage`, `restorecon`, `newrole`
Startup utilities	`bootloader`, `initrc`, `init`, `local_login`
File tools	`dpkg_script`, `dpkg`, `rpm`, `mount`, `fsadm`
Network utilities	`iptables`, `sshd`, `remote_login`, `NetworkManager`
Auditing, logging services	`logrotate`, `klogd`, `auditd`, `auditctl`
Hardware, device management	`hald`, `dmidecode`, `udev`, `kudzu`
Miscellaneous services	`passwd`, `tmpreaper`, `insmod`, `getty`, `consoletype`, `pam_console`

Figure 1: A list of trusted applications in the SELinux release for Fedora Core 5 using mls-strict policy version 20.

files). In the following code (written in Jif syntax), data is read from the keyboard on line 1 and properly stored in a variable labeled with Alice's policy. In line 2, the label on `leak` can be inferred as `{alice:}`. Then a file is opened to write out configuration information (which is publicly readable). A leak occurs, however, when the program attempts to write Alice's data out to the configuration file. This code also contains a security violation in line 10, because statistics, which Alice should not be able to access, are printed to the screen. The typechecker would flag these errors and prevent this program from compiling.

```
1.  String{alice:} secret = stdin.read();
2.  String leak = secret;
3.  FileOutputStream[config] conf =
4.    Runtime.openFileWrite("tool.conf",{config:});
5.  conf.write(leak);
6.  FileInputStream[state] statsFile =
7.    Runtime.openFileRead("stats.dat",{state:});
8.  String stats = statsFile.readLine();
9.  if (stats.split[0].equals("bob"))
10.   stdout.write(stats);
```

Returning to Figure 2, we can presume that objects stored in the system are already labeled (with an OS label, L_{os}, for example), but we still need an OS API to get the labels and provide them to the application. Additionally, this must connect into a language-provided API to translate these labels into labels that the application can enforce (L_{ap}). This must be a carefully controlled interface so that the labels cannot be manipulated or spoofed. Finally, the information flows that the application will enforce must *comply* with the information flows enforced in the operating system. In the example, if the operating system policy were to state that $L'_{os} \leq L_{os}$ (L' is less confidential than L), but the application policy still has $L_{ap} \leq L'_{ap}$ (L is less confidential than L'), then the application would violate the operating system's policy.

Compliance testing is complicated by the mismatches between the lattices used by the operating system and those used by the application for enforcing information flow. Firstly, there may be principals in each lattice that are not found in the other. These cannot merely be removed, because they might connect shared principals and be involved in information flows. Secondly, there may be a mismatch in the kinds and granularity of permissions that the OS handles (set attribute, open, link, delete, read, etc.) compared to the application. Finally, the application policy could be more restrictive than the OS policy, but the reverse should not be true.

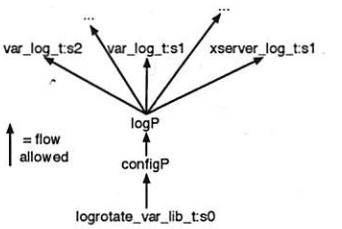


Figure 3: The lattice of principals describing all possible flows for `logrotate`. The two basic levels are logs and configuration files. The data in configuration files affect the logs but the reverse should not be true. Also, the logs should not be able to flow into each other.

For example, consider the lattice we constructed for `logrotate` in Figure 3. `logrotate` only needs to handle two kinds of files—the log files and the configuration and state files. Furthermore, the log files should be disjoint from each other and more sensitive than the configuration and state files. In this lattice, `configP` and `logP` do not have corresponding principals in the operating system. Also, we can see that this policy is more restrictive than the OS policy (notice that `var_log_t:s1` is normally be able to flow into `var_log_t:s2` according to OS policy, but not in this lattice). Also, this policy does not capture all possible OS principals. Finally, this policy only describes basic information flows–read and write.

We identify the following tasks. (1) We need a mechanism by which an application can prove that its information flow enforcement does not violate the system information flow policy. Both Jif/Pol (JP) applications and SELinux express the information flow policies that they are enforcing, so we need an approach to compliance testing the application policy against the system policy. This must include some control of application-level declassification preventing both unacceptable declassification filters and also the overuse of applications with declassifying filters. (2) We need mechanisms for the appli-

cation to determine the label of its input channels necessary to enforce information flow. If JP applications cannot distinguish between secret and public inputs, it must label them all secret to enforce information flow requirements, thus impacting usability. (3) The system must be able to determine the label of all JP application outputs. Again, the lack of an accurate label would either result in overly conservative enforcement (i.e., the application may only send secrets) or possible vulnerabilities (i.e., the application sends a secret to a public entity).

To summarize, these considerations motivate the following concrete problems:

1. How can we pass operating system resources along with their labels into an application?
2. How can we pass application data along with their labels out into the operating system?
3. How can we be sure that the application will faithfully enforce the operating system's policy on these labels?

With the solution of these problems, we have a guarantee of system information flow enforcement, based on reconciliation of information flow enforcement and accurate communication of information flow labels between application and system layers. In the next section, we provide an architecture that solves these problems. In Section 4, we give the details of our implementation of this architecture for Jif and SELinux.

3 Architecture

In this section, we provide a general architecture for solving the problems described in Section 2. Namely, we describe the necessary steps for ensuring that a security-typed application can handle data with a range of security levels and still enforce the information flow goals of the OS. Note that this architecture is independent of any particular language or OS. We describe, in general, the features that are required for our approach. In Section 4, we will describe our implementation of this architecture for Jif and SELinux.

3.1 Process steps

We begin with a description of the overall process and then focus on the details of various steps in the subsequent subsections. Our five process steps are illustrated in Figure 4.

0) Initial state The OS must have a MAC policy implementing some information flow security goals. We focus on SELinux in this paper, but this could include other high assurance operating systems such as Trusted-Solaris or TrustedBSD. The key is that there must be an explicit MAC policy that is accessible to a system daemon for analysis of confidentiality policies. (Here we focus on ensuring confidentiality, but other information flow goals could also be examined.)

1) Program secure application An application developer provides the bytecode for a security-typed application along with a policy template that can be specialized by the user for a particular operating system configuration. We have used Jif/Pol in this paper, but the concepts extend to any security-typed language. A key point is that the language must provide a policy system such that each application will have an explicit policy that can be analyzed by a system daemon to understand the security lattice and declassifiers the application uses. We discuss this further in Section 3.2.

2) Specialize application policy Although a program will be developed with some basic security goals in mind, the application policy may be customized for different users running on different systems. This is especially important because the application policy must make connections to operating system label names which may not be the same from system to system. Of course, a reference policy should always be provided by the developer which should run on a default system configuration. The reference policy also serves as a template for customization to a customized OS. We discuss this further in Section 3.3.

3) Invoke service In an MLS environment, a user may have the authority to run at various security levels, but typically only logs in at one level at a time. In our approach, when he desires to run an application with a range of levels, he must first invoke an operating system service to check the application for compliance with operating system security goals. There must be no way to subvert this, i.e. to run the application without allowing the system first to perform the necessary checks. This should be enforced in the system policy.

The operating system service performs checks based on four inputs: the system policy, the object code for the application, the application policy and the desired range of levels. We discuss this further in Sections 3.4 & 3.5.

4) Run application If all the checks succeed, the operating system service may launch the application at the requested security level range.

3.2 Programming infrastructure

To address the last two problems listed in Section 2, namely that operating system resources—both inputs and outputs—must be labeled properly in the application, operating system and language APIs are necessary. First, the operating system API must offer procedures for an application to get labels on files, sockets and other OS resources. It must also be possible for the application to set labels on resources when they are created by the application. Secondly, the security-typed language API

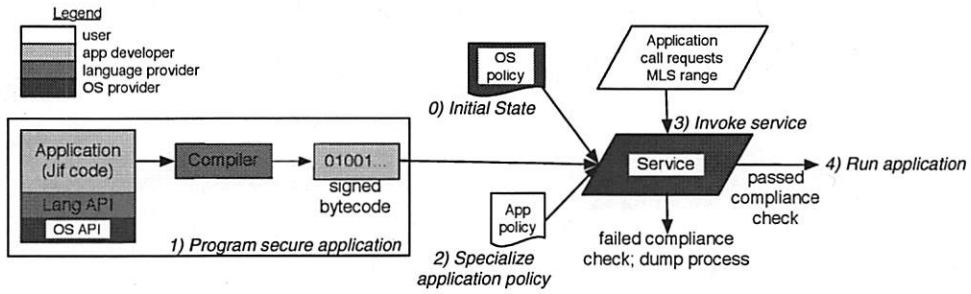


Figure 4: The process for executing an application with range of MLS privileges consists of 5 steps. The steps are performed and components are provided by different entities as shown by the different colors.

must supply procedures for getting and creating operating system resources. The primary concern is that these API abstractions provide the *only* way to access operating system resources. One solution, the one we use, is to provide a single way of creating new, or opening existing resources. With this approach, when an application's data structure is mapped onto a system resource, the internal label assigned to the data structure can be checked to correspond with the external label on the system resource. Thereafter, throughout all possible program executions, the normal security-type analysis provided by security-typed languages can ensure that that label is never violated.

3.3 Specializing application policy

Our approach assumes that a developer will construct applications that enforce some security goals. For example, a program variable into which a secret password will be read should be labeled differently from a variable that will contain public information. This must be part of the program code. The *meanings* of these labels are established in a high-level application policy external to the program code, however, and configured according to user preferences and system policy. For example, the public information could be treated just as secretly as the password if the user desires. Furthermore, the application developer will not know the names of security labels on the user's operating system; these must be configured by the user in light of the operating system policy. Another consideration is that a user may prefer not to use certain declassifiers in a particular application; this should be customizable in the application policy. Once the application policy has been specialized, the policy and application can be passed to an operating system service for compliance checking and execution.

3.4 Verifying run requests

The operating system service must be on the critical path for running any security-typed application, because it will ensure that all three requirements listed in Section 2 are met. Namely, it will ensure that 1) the labels passed into and 2) out of the application are consistent with the operating system labels and it will ensure that 3) the application will enforce the operating system's policy throughout its execution. To do this, the service must make four checks: (1) the application's information flow policy must be provably compliant with the operating system policy, (2) the code should be verified as having been compiled by a proper security-typed language compiler, (3) the declassification filters required by the application, if any, must be acceptable for the operating system and (4) from a global view, there is no suspicious behavior in running this trusted process that would appear to be covert channels (such as forking dozens of processes which might each leak a small amount of information by their existence and through declassifiers).

The key requirement here is the first, compliance testing, which is discussed in more detail below. The other three requirements are more general or more ad hoc—there are no general solutions so lots of ad hoc ones are possible. They remain ongoing areas of research. We discuss some preliminary approaches in Section 4.2.

Note that because the service itself is trusted to handle multiple security levels of data, it should either be written in a security-typed language and bootstrapped into place, or it should be small enough to be verified by hand.

3.5 Compliance analysis

A trusted application is given some flexibility to handle information in ways that a normal application could not. Before granting such privileges, however, the operating system should check to be sure that the application will not abuse them. In other words, the application policy should *comply* with the operating system policy.

> An application is said to be *compliant* if it introduces no information flows that violate the policy of the operating system in which it is running.

For a security-typed application *all* possible information flows can be determined based solely on the

high-level delegation policy, modulo some declassifications [13]. The operating system need only check the compliance of flows that are relevant to its own principals.

The compliance analysis between a security-typed application with high-level policy and a MAC-based operating system with static policy consists of three steps. (1) Convert the application policy and operating system policy into a form in which they can be compared. (2) Determine which security levels are shared between the operating system and application. For each of the security levels, collect all allowed flows for the application. (3) Compare these to the flows allowed by the operating system. If there are strictly more flows allowed by the operating system with respect to shared security levels, then the application can be declared compliant and can be safely executed.

This problem contains several challenges. One is that the OS may contain security levels not used in the application and the application may contain security levels not used in the OS. Another is that the OS and application may have a mismatch in the granularity of permissions. Also, either policy could be quite large and unwieldly, making analysis slow or even intractable. These were all problems we had to solve when implementing this analysis for Jif and SELinux. We describe our implementation in Section 4.3.

4 Implementation

For our implementation, we use the Security Enhanced Linux (SELinux) operating system [25] provided as part of the mainline Linux kernel. To build our secure applications, we used the most mature security-typed language, the Java + Information Flow (Jif) language [23], augmented with our policy system (Jif/Pol). Jif is the only security-typed language with an infrastructure that was robust enough to be expanded to handle the kinds of system calls that were necessary for interacting with SELinux. Because we were focused primarily on confidentiality, it was sufficient for us to use Jif v. 2.0.1 (v. 3.0 adds integrity to the security labels in Jif and is a target for our future work).

Our implementation consists of three major endeavors. First, we extended the Runtime infrastructure of the Jif compiler with an interface to SELinux kernel 2.6.16 for getting and setting SELinux security contexts on network sockets and files. In order to make this configurable we added some primitives to the Jif/Pol language and implemented the changes in the Jif/Pol policy compiler. Second, we constructed the Service for Inspecting and Executing Security-Typed Applications (SIESTA). This includes a system daemon along with an interface that can be run by the user; both were written in C. It also includes a policy compliance checker which was written in XSB Prolog. Thirdly, we have utilized this infrastructure to build and test two demonstrative applications: `logrotate` and JPmail.

4.1 Extensions to the Jif Runtime

The basic paradigm in Jif for labeling operating system resources is to parameterize the resource stream with a label and pass that label into the proper method of the `Runtime` class when opening or creating the resource. The `Runtime` method then checks to ensure that the label passed in by the application is acceptable (not too high, not too low) for the resource being requested. For example, the following code gets the standard output stream and attempts to leak a secret.

```
// user is a principal passed in through main(...)
Runtime[user] rt = Runtime[user].getRuntime();
final label{} lb = new label{user:};
PrintStream[lb] stdout = rt.stdout(lb);
int{high:} secret = ...;
stdout.println(secret);
```

Jif ensures (1) that the `Runtime` class, which is instantiated by `getRuntime()`, is parameterized only by the user who executed this program (for SELinux this would be the security context of the program). Jif also ensures that when creating a stream for `stdout`, that the stream is parameterized by a label which is (2) equivalent to the label passed as a parameter to `rt.stdout()` and (3) no more secure than the label on the `Runtime` class. This corresponds to the notion that we should be able to print public data or user data to the user terminal, but nothing more secret than this.

Following this paradigm, we extended the Jif `Runtime` to handle labeled IPsec sockets (both client and server sockets) and labeled files. The basic concept is straight-forward: we provided `openSocket`, `openServerSocket`, `openFileWrite` and `openFileRead` methods which ensure that the streams attached to these operating system resources are properly labeled within the Jif application. The details of this implementation required some additional work, however, in order to properly interface with the SELinux API for getting and setting security contexts on sockets and files.

The code for implementing the socket interface was particularly challenging, because of the way labeled IPsec handles SELinux security contexts and IPsec security associations (SAs). Namely, in order to provide the proper cryptographic protocols for a particular socket, it is necessary that the application first creates the socket, then assigns the proper security context and then attempts to make a connection (it must occur strictly in that order). At that point, the IPsec subsystem attempts to establish an SA for the given security context, local host and port number and remote host and port number.

The problem is that the standard Java Socket API (on which we must build for Jif) does not distinguish between creating a socket file descriptor and attempting to make a connection. Consequently, we had to extend the Socket class to implement our own `SelinuxSocket` and `SelinuxServerSocket`. The constructors for our new classes can take a security context. Then, when the socket attempts to establish a connection, a shim is inserted between socket creation and socket connection that calls into the SELinux API to change the socket's security context. After connection, the `Runtime` class ensures that the SA retrieved for the socket corresponds to the security context that it was set to.

The rest of the code in `Runtime.openSocket(...)` follows the model of `stdout(...)`, checking to ensure that the label which is attached to the Jif `Socket` object corresponds to the security context attached to the operating system resource. The difference is that because sockets are always two-way in Java, the label must be equivalent (neither higher security nor lower security) to the SELinux security context.

Extending the Jif policy system Because Jif and SELinux use different kinds of labels and principals, we needed to make some connection between them. We handled this by extending the Jif/Pol policy language with an operator, $[\cdot]$, to signify an operating system label and also wild cards to match a series of labels.

Consider the lattice for `logrotate` in Figure 3. The SELinux principal `logrotate_var_lib_t:s0`[3] is at a lower secrecy level than the Jif principal `configP`. With our new policy syntax, we can express this relationship as `[.*:.*:logrotate_var_lib_t:s0] -> configP`. The Jif Runtime does not create principals in advance to correspond with every operating system principal; it only creates them as needed (e.g. when a file labeled with that principal is opened). Furthermore, when they are created they are assumed to be unrelated (incomparable) to any other principals in the lattice. Thus, the effect of the policy statement given above is that a hook is inserted into the Runtime to watch for any principals matching this wildcard and when one is created, it will be properly placed in the lattice.

When the relationship is reversed and a Jif principal must be lower in the security lattice than an SELinux principal, the relationship is stored in the Jif principal at the time of its creation (the beginning of the program). For example, if we have the policy statement `logP -> [.*:.*:.*:.*]`, placing the Jif principal `logP` lower in the lattice than any SELinux principal. Note that this policy statement does not presume anything about the relationship between different SELinux principals that match the same wildcard—only that each of them is higher than `logP`.

To summarize, the lattice policy we used for `logrotate` is small and concise (it is the lattice that was illustrated in Figure 3):

```
[.*:.*:logrotate_var_lib_t:s0] -> configP;
configP -> logP;
logP -> [.*:.*:.*:.*];
```

4.2 SIESTA

The Service for Inspecting and Executing Security-Typed Applications (SIESTA) consists of two parts: a service interface and a system daemon, as shown in Figure 5. The service interface takes two inputs from the user—the security-typed application to be executed and the desired MLS range at which it should be executed. The service interface calls a long-running system daemon to carry out the checks described in Section 3.4. If everything is acceptable, SIESTA proceeds to execute the Jif application with the special MLS privileges. In the following, we describe SIESTA's operation in more detail.

The SIESTA service interface starts running with the same MLS level as the process that called it. Running the interface also causes an SELinux transition into the `siesta_t` domain which limits the process's functionality to communicating with the daemon and forking a new trusted Jif application. The communication between the interface and the daemon is supported by IPC mechanisms plus security functions, creating what we called a SIESTA channel. Furthermore, supported by OS policy we make the `siesta_t` domain the sole gateway for executing Jif applications in a domain with special MLS privileges. This guarantees that the user cannot directly run a Jif application with special privileges unless it has first been checked by SIESTA. The logic in the service interface is quite simple, deferring the complex considerations to the system daemon.

First, the SIESTA system daemon is responsible for ensuring that the Jif application it has been given is trustworthy. It must first ensure the "Jif-ness" of the application bytecode archive (jar). Doing this in a general way is really an orthogonal issue and a research topic in itself, so here we take a straight-forward approach and just validate a jar signature against a potential list of third-party, trusted Jif compiler signatures.

Once the Jif-ness of the application has been established, the policy jar that has been passed to the daemon must be checked for compliance against the system policy. The jar contains the Jif-compiled policy files (principals and policy store) that will govern the application while it executes. It also contains a manifest of the policy from which it was built. The policy compliance check is described in more detail in Section 4.3.

Thirdly, the SIESTA daemon ensures that the declassifiers used by the application are acceptable to the operating system. Although there are no general solutions to

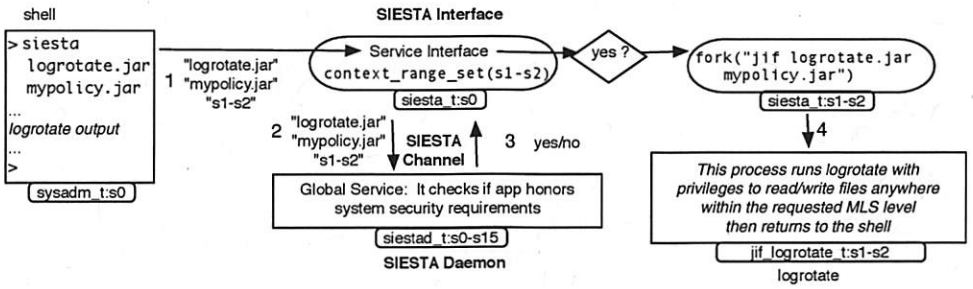


Figure 5: SIESTA: a service which securely validates and executes trustworthy security-typed applications.

this problem, we provide a preliminary approach. Before running an application, a compliance algorithm should check the declassifiers used by the application against a white-list or black-list of declassifiers. For example, standard military procedure prohibits the use of DES for protecting secret data. This can be easily checked in the application policy and applications that violate such requirements can be prevented from executing. As more security-typed applications are used, a list of trusted declassifiers can be established and become a more natural part of the operating system policy. Also, some applications, like `logrotate`, don't need any declassifiers at all. Other approaches could also be feasible here, taking advantage of ongoing research in quantifying the information leaked through declassification [28]. We discuss this further in Section 5.1.

Lastly, although we do not attempt to implement any particular policy for eliminating the covert channels which could be created through the execution of hundreds of these security-typed applications, we provide hooks that could be used as such policies are developed.

The SIESTA daemon must be executed by a system administrator prior to executing any security-typed applications. It must run with a full range of MLS privileges so that it can handle security-typed applications of all sensitivities. At the same time, it can be limited to a fairly constrained functionality, because it only needa to read from files and communicate with the SIESTA service interface via IPC.

4.3 Compliance analysis

There are a few key challenges in attempting to determine compliance between Jif policy and SELinux policy. The foremost challenge is in the semantic difference between Jif's information flow lattice and SELinux's MLS constraints. Although SELinux claims to have an MLS policy (which normally means a "no read-up", no write-down" lattice-based policy), the so-called "MLS" extension is really a richer policy language which can be used to implement MLS, but can also implement more general policy goals. The second challenge lies in the difference in granularity between Jif policy and SELinux policy. While SELinux policy distinguishes various operations and resource types (the policy for setting the attribute on a file could be different from writing to a socket, for example), Jif policy gives a more comprehensive view of all information flows in an application. The third challenge lies in the size of the SELinux policy for a whole operating system. The standard, complete operating system policy consists of well over 20,000 policy statements.

For the first challenge, analyzing policy compliance would have been a straight-forward lattice comparison if not for the generality of the SELinux MLS policy. Thus, some policy analysis tools are needed to determine what information flow goals are implemented by the operating system and whether they are compatible with the information flows in the application we are seeking to execute. Although several SELinux policy analyzers exist, none were suitable for our purposes, because none handles the new SELinux MLS extensions which were our primary concern. Consequently, we developed our own policy analysis tools for SELinux MLS policy, inspired by the policy analysis engine, PAL [30]. Our tool determines what information flows are allowed between MLS levels. We describe this analysis in more detail elsewhere, including a formal consideration of correctness [14]. For this work, we extended and utilized this tool to compare the flows allowed in a Jif application to the flows allowed in the host operating system.

The second challenge is that in order to compare the operating system policy and the application policy, they must be in a comparable form. Since SELinux policy is more general than Jif policy, we translate our Jif policy into an SELinux policy. This also allows us to reuse our policy analyzer for both policies.

For example, consider the Jif policy in Figure 6 in the box labeled `app-policy.jifpol`. This policy says that the Jif program has access to operating system files and network sockets. Also, it allows data to flow from `pub` to `siis` to `sec`. Furthermore, the policy states that `pub` is equivalent to the security level `s0` and `sec` is equivalent to the security level `s1`, while `siis` has no corresponding

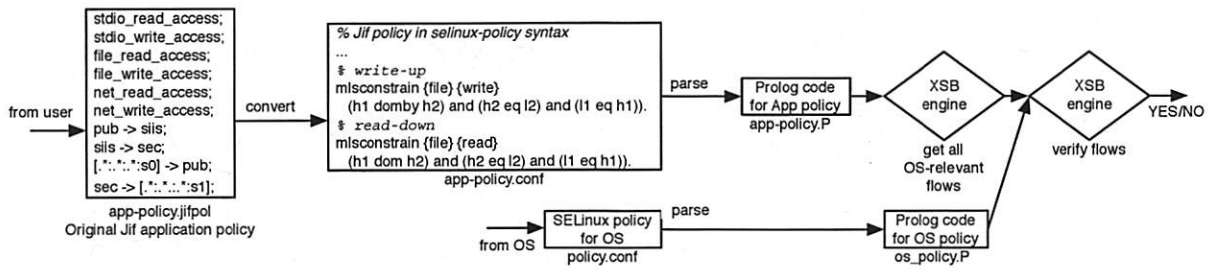


Figure 6: The compliance testing process.

identity in the operating system.

Given this policy, the contents of an operating system file at level `s0` could be read into the application at level `sec` (through a read-down) and then written out to a file at level `s1`. This constitutes a flow from `s0` to `s1`. This flow must then be checked against the operating system's policy to determine if it is an allowed flow. Note that although `siis` has no corresponding principal in the operating system, it cannot simply be ignored, because it could be involved (as in this case) in a flow between two operating system principals. At the same time, the flow from pub to `siis` need not be checked against the operating system policy, because the `siis` principal does not correspond to an OS principal. Only when both endpoints of a flow have corresponding OS principals does the flow need to be checked against the OS policy.

Next, if the Jif application asks for `file_read_access` and `file_write_access`, we then add, respectively, read-down rules and write-up rules for file access, giving an SELinux-style policy as shown in the box, `app-policy.conf`. We add similar rules for user I/O if `stdio_read_access` and/or `stdio_write_access` are set and for sockets if `net_read_access` and/or `net_write_access` are set in the Jif policy.[4]

The third challenge we faced was the magnitude of the operating system policy, which threatened to make the analysis intractable. We are able to manage this in several ways. Firstly, once the policy has been compiled into Prolog, it need not be compiled again. Furthermore, XSB Prolog has some efficient methods of storing the policy, using tabling, which improve performance for analysis. Most importantly, however, we are able to radically reduce the analysis of the operating system policy by first analyzing the application policy. Because we are only interested in verifying that the flows allowed by the application are also acceptable to the operating system, we don't need to check all operating system flows—just the ones that intersect with the application.

4.4 Sample applications

We have implemented two sample applications in order to demonstrate the range of functionality provided by our architecture. The first is `logrotate` which demonstrates proper labeling of files and tracking of information flows from multi-leveled files handled within the same application. This is an example of a secure implementation of an operating system tool and demonstrates features that would be common to many of the utilities listed in Figure 1. The second application is larger and more complex—a modification of the JPmail email client [12]. For this work, we migrate this client from using a PKI for achieving end-to-end confidentiality to using the SIESTA infrastructure with labeled IPsec.

4.4.1 `logrotate`

The `logrotate` utility is regularly executed by cron to gradually phase out old log files. The utility rotates each set of log files based on some configuration. For example, the `messages` log is renamed to `messages.1`, `messages.1` to `messages.2`, etc. The configuration specifies which logs to rotate and each log has a `rotate` attribute indicating how many back logs to save. The full version of this utility can also run scripts, compress logs and send emails. We did not implement these additional features, but chose to focus on the essential functionality of log rotation.

The `logrotate` program handles a variety of sensitive information flows (an example lattice is shown in Figure 3 and the lattice policy is given in Section 4.1). It handles two files which are (typically) publicly readable: a configuration file and a state file. It handles various other log files at various security levels, creating and modifying the files as needed in order to rotate and delete logs according to their particular configurations. The data in the log files is more or less secret depending on the nature of the log. For example, the logs for X Windows and `wtmp`[5] are usually publicly readable. Other logs such as `secure` or `maillog` are more secret due to their contents. On the other hand, the attributes of the log files (e.g. seeing that they exist, getting their names, reading their last date of modification, etc.) are public.

In order to rotate logs, the program needs to read configuration information and state information and based on that, the logs themselves are renamed. This effectively passes information from the configuration files into

the log files (it is clear from looking at the directory, for example, what the `rotate` attribute is for each log—usually it is the highest filename extension, like the 4 in `messages.4`). Thus, in order for our application to function properly, the level of the configuration data must be lower or equal in the lattice, i.e. less secret, compared to the level of the log data. On the other hand, we do not want to leak log data into the configuration file (since it is often publicly readable) or into other log files. In fact, our Jif application verifies that this policy can be upheld: not even small bits of information released by control flows are leaked from the log files into the configuration files and not even a single declassifier is needed to implement this system utility.

4.4.2 Email client

The JPmail application [12] is an email client built in Jif 2.0.1, using our Jif/Pol policy framework [13], which enforces information flow control on emails according to a given Jif policy. When we built JPmail, it was the first real-world application built in Jif. Previously, in order to maintain information flow control, JPmail utilized encrypting declassifiers to send out email on public networks. By utilizing labeled IPsec sockets and trusting the operating system to handle distributed security (i.e., the MAC OS security ensures that emails are not leaked from intermediate or destination servers), we were able to remove the cryptographic infrastructure from JPmail and significantly simplify the code. Furthermore, we were able to extend our mail client to handle communication with mail servers at multiple security levels within a single process.

While this application serves to demonstrate the usage of client sockets, the real significance of this application is mainly in its size and complexity. It is the largest existing Jif application and so it gives us some insight on the difficulty of augmenting a realistic application to work with SIESTA. In this vein, we were gratified to discover how much cleaner and simpler the code became when it could trust the operating system to handle security concerns over its resources.

Also significant about this application is its use of declassifiers. This is due to the fact that it gets user input for all levels of email accounts from the same terminal window. The application logic handles the proper downgrading of input when responding to a public as opposed to a secret email. Another, minimal source of leakage is through an implicit flow caused by handling both public and secret email accounts in the same user interface loop. This small flow that normally occurs when a single user interface is used for multi-level inputs is handled with a declassifier. The declassifier and its use in the code must be determined to be safe for the email client and then it is included among the white-list of declassifiers in the operating system policy.

5 Discussion and evaluation

5.1 Declassification

Strict information-flow policies are too strict for some applications. This necessitates slight relaxations of the policy through controlled "escape hatches". There has been a great deal of consideration about declassification in the language-based security community [28]. We have added our own modest work, called *trusted declassification*, to this collection [13] with the greatest advantage being its practicality and the way it exposes declassifiers through a high-level policy. This exposure of declassifiers is key for our compliance analysis. The policies it allows are similar to recent work on integrity policies for generating Clark-Wilson Lite models [31] of security.

The key is that all declassifying filters (aka declassifiers) must be declared in the high-level policy, indicating what information flows they can be used for. For example, in order for an application to expose secret data (labeled `{sec:}`) to the public (labeled `{pub:}`) after encrypting it with AES, the application's policy must contain the statement, `sec allows crypto.AES(pub);`. Otherwise, when the application tries to use the declassifier at runtime, the policy check fails and an exception is thrown. Thus, an application's code may contain potentially many declassifiers, but only those which are explicitly allowed in the policy can be used at runtime (the compiler ensures that all necessary runtime checks are present in the application before it generates the object code).

5.2 Performance

In this section we consider performance costs associated with the approach outlined in the preceding sections. We stress the preliminary nature of the implementation, experiments, and test-bed. Because of the unique and cross-cutting nature of these experiments, it is highly difficult to isolate performance cost (simultaneously at the application, OS and network layers). Experimental error is caused by interactions between the OS (process scheduling, interrupts), network delays, Java (garbage collection and dynamic class loading), and other system services and applications (process interference). Hence, we focused our initial experimentation toward obtaining a broad performance characterization of the design, leaving more precise evaluation and the invention of apparatus to achieve it to future work.

We study the overheads associated with information flow controls at the application (Jif) level. We compile the Jif programs using Ahead-of-Time (AOT) compilation with the `gcj` compiler v. 4.1.1 with `classpath` 0.92 [29]. We examine two operations, *a*) *logrotate* which renames up to four log files per set and as many

Operation	Configuration	Median	Mean	σ
logrotate	C	7.923501	7.943820	0.133496
logrotate	Jif	13.949643	13.925600	0.122477
send	C	17.825400	21.834692	12.163714
send	Jif	12.522900	15.620364	10.705158
SIESTA	compliance	241.060289	252.830086	25.025038
SIESTA	cached	31.639957	32.368633	3.353408

Table 1: Time (ms) to send a 10KB email in both Jif and C, time (ms) to perform one rotatation of 50MB of log files and time (ms) to start up the Jif process using SIESTA (includes Jif-ness validation and compliance checking).

sets as requested, and *b*) *send* which sends a single email from the client to the server. For `logrotate`, we compare our Jif application with the latest C version 2.7.1, using only minimal features of the applications. For sending mail, we compare a custom C-based MTA application with JPmail with IPsec enabled. A 3DES ESP with MD5-integrity policy was used in all IPsec-enabled tests. The tests were run between two identical 3GHz Intel hosts running SELinux Kernel version 2.6.16 with 1GB memory on a lightly loaded 100MBit network. All experiments were repeated 100 times.

Table 1 provides macrobenchmarks for the different operations and configurations when sending a single 10KB email and when rotating forty log files totalling 50MB. For sending email, the overheads of the system/approach are relatively small: in all cases the average execution time is less than 25 milliseconds, and in many cases significantly less. In general, costs are in line with unprotected systems, which indicates operations such as these are likely to be unaffected by the additional security infrastructure.

5.2.1 Sample applications

In the case of `logrotate`, the Jif application consistently runs 2x slower than the C version. We tested the two programs with various log files of different total sizes. Since there is no inspection of the contents of the files, the size is the determining factor. The displayed result is for a standard complement of log files totalling 50MB. For this utility, the decrease in speed is inconsequential. The `logrotate` application is generally a cron job that only runs daily or even weekly. Additionally, our Jif code is entirely unoptimized and could be improved.

A further comparison of the Jif and C applications shows, interestingly, that Jif is faster, on average, for email sends, although there is a significant variance for both C and Jif. We found the Jif functions represented a vanishingly small amount of overhead in this case. Further investigation revealed that a significant portion of the additional costs observed in the Jif application are due to delays in the use of Java network APIs. The C program is slower for sends because of an implementation artifact: JPmail does a less graceful exit with the server, whereas the C program waits for the server finalization.

5.2.2 Compliance testing

For SIESTA, the overheads are constant and small. The policy compliance check requires a call into the XSB prolog interpreter but executes relatively efficiently, requiring only 15.512256 ms on average. 5.577328 ms of this time is spent in loading the policy (both for the application and the OS) while 8.902952 ms is spent doing the flow checks. XSB prolog is highly optimized and the prolog source files can be compiled for greater efficiency. The majority of time is spent in signature validation for the jar file that is being loaded.

Fortunately, this validation process (checking Jif-ness and checking compliance) is a one-time cost when first verifying the compliance of the application and its policy. This process only needs to be repeated if the application changes (for a new version), if the application policy changes or if the OS system policy changes, all of which should be rare events. Otherwise, the service may compile the jar file together with its policy into a binary executable and the hash of the binary can be cached for future executions. Checking the hash is almost an order of magnitude faster than validating the jar signature and checking compliance (32.368633 ms vs. 252.830086 ms averaged over 100 runs).

5.3 Practicality/usability

What we have implemented is a prototype using Jif as a basis for trust when constructing trusted applications for a secure operating system. The guarantees that we are capitalizing on are not specific to Jif, but are part of the static type analysis that Jif implements for security types. Jif has some problems. For example, Jif is built on Java and requires that the JVM be loaded. This may not be desirable for some applications. Furthermore, the JVM introduces a large amount of code into the trusted computing base. Fortunately, this is not an inherent limitation to our approach because the security-type analysis we depend on is orthogonal to the JVM (and any security goals it may implement).

Another limitation is that Jif is not easy to program in. Some of this is inherent in the fact that it must be thorough about checking all information flows. For example, it requires all exceptions to be handled, including `NullPointerExceptions`, and it tracks implicit, con-

trol flows, which can be difficult for a programmer to follow. On the other hand, we believe that much of the burden can be alleviated through some semi-automated labeling and through the development of other tools and programming patterns [1, 12]. We are currently investigating these avenues. In the meantime, it is also important that the development of these tools be guided by practical experience and some knowledge of how they can be deployed. That is what we have described in this paper.

6 Related work

This research considers the intersection of two areas of related work: 1) secure systems development, particularly mandatory access controls and 2) language-based security, particularly security-typed languages.

Mandatory Access Controls The foundation of our OS work comes from the Flask architecture [34], which has been integrated into Linux through the Linux Security Module (LSM), giving Security Enhanced (SE) Linux [32]. This is now being shipped as part of the mainstream kernel in the 2.6 series and turned on by default in Redhat distributions since Fedora Core 5. Other work in operating systems with MAC security include Trusted Solaris [21], Solaris Trusted Extensions [20], TrustedBSD [9] and SEDarwin [35].

MAC Operating Systems require all subjects and objects are labeled and all security-sensitive operations are hooked with runtime checks. These checks query a previously configured security policy to determine whether the operation is allowed, based on the subject and object labels.

These policies have been used to implement various high-level information flow properties across a distributed system. Recent research has shown that this basic mediation (called Type Enforcement or TE) can be used to enforce integrity constraints on data [15]. More recently added multi-level security (MLS) labels can be used to enforce confidentiality [11]. A major limitation in all this work is that it only observes security-sensitive operations from outside of applications; it cannot peer into application code to catch data leaks.

Security-typed languages Security-typed languages have their heritage in the information flow policies of Bell & LaPadula [3] with extensions to lattices as described in [7]. This led to the concept of *noninterference* [10], in which modifications of high security data cannot be observed in any way through low security data. Thus, two execution traces with different high security inputs yield the same low security outputs. In their seminal work, Volpano, Smith and Irvine [36], showed how these information flow policies could be encoded into types and noninterference could be determined through a type analysis. The first language to implement this was a variant of Java, called Java + Information Flow (Jif) [22], which uses labels based on the Decentralized Label Model (DLM) [24].

Jif remains the most mature security-typed language, although there is much activity in the field (see a recent survey for more details [27]). Other security-typed language projects include functional, [26], assembly [5] and web scripting [17] languages, as well as language features for multiple threads of execution [33] and distributed systems [18]. Other recent work has studied *integrity* information flow [4, 6] in the context of replication and partitioning [37]. Much of the work in security-typed languages has been theoretical, but some recent work demonstrated that these languages can also produce real-world applications [12].

A recent project, called GIFT [16], implements a more general, but less rigorous approach to tracking data flows within C applications. This language framework may be another useful target for our architecture.

7 Conclusion

In this paper, we have described an important problem in secure systems development, namely the inability of an OS-level reference monitor to look inside a multi-level application. We have provided an architecture to solve this problem by using security-typed languages to track secure data flows within applications. We implemented this architecture for the security-typed language, Jif, and the MAC operating system, SELinux. Through the applications, `logrotate` and JPmail, we showed that our approach is secure, flexible and efficient.

Acknowledgements

A special thanks to Steve Chong for his tireless help in providing insight about the inner workings of Jif and to Mike Hicks for his numerous editorial comments. This work was supported in part by NSF grant CCF-0524132, "Flexible, Decentralized Information-flow Control for Dynamic Environments" and NSF grant CNS-0627551, "CT-IS: Shamon: Systems Approaches for Constructing Distributed Trust".

Notes

[1]To be complete, in addition to these five sensitivity levels there are also category sets, but we leave them out here for simplicity of presentation, as they do not add to the technical complexity.

[2]In SELinux, "MLS" has a broad meaning with the names and semantics being drawn from a general policy. We consider the implications of this more in Section 4.3. The meanings of all type and MLS level names are also defined in the policy, but typically s0 is most public and s15 is most secret.

[3]Recall that logrotate_var_lib_t:s0 is an abbreviation for system_u:object_r:logrotate_var_lib_t:s0.

[4]For brevity and clarity of presentation, we have given a truncated version of the policy, but to be complete, our implementation includes

all the write-like and read-like operations necessary. For the same reason, although our implementation also handles category sets, we forego a discussion of that here.

[5] wtmp is queried by the UNIX command last.

References

[1] A. Askarov and A. Sabelfeld. Secure Implementation of Cryptographic Protocols: A Case Study of Mutual Distrust. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS '05)*, Milan, Italy, September 2005.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, October 2003.

[3] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, March 1976.

[4] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[5] E. Bonelli, A. Compagnoni, and R. Medel. Non-interference for a typed assembly language. In *Proceedings of the LICS'05 Affiliated Workshop on Foundations of Computer Security (FCS)*. IEEE Computer Society Press, 2005.

[6] S. Chong and A. Myers. Decentralized robustness. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)*, Venice, Italy, July 2006.

[7] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.

[8] S.W. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. VMWare, Inc., October 1998. US Patent No. 6397242.

[9] FreeBSD Foundation. SEBSD: Port of SELinux FLASK and type enforcement to TrustedBSD. `http://www.trustedbsd.org/sebsd.html`.

[10] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[11] Chad Hanson. Selinux and mls: Putting the pieces together. In *Proceedings of the 2nd Annual SELinux Symposium*, 2006.

[12] B. Hicks, K. Ahmadizadeh, and P. McDaniel. From Languages to Systems: Understanding Practical Application Development in Security-typed Languages. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC 2006)*, Miami, FL, December 11-15 2006.

[13] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification: High-level policy for a security-typed language. In *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '06)*, Ottawa, Canada, June 10 2006. ACM Press.

[14] Boniface Hicks, Sandra Rueda, Luke St. Clair, Trent Jaeger, and Patrick McDaniel. A logical specification and analysis for SELinux MLS policy. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, Antipolis, France, June 2007.

[15] T. Jaeger, A. Edwards, and X. Zhang. Policy management using access control spaces. *ACM Trans. Inf. Syst. Secur.*, 6(3):327–364, 2003.

[16] L. Lam and T. Chiuch. A general dynamic information flow tracking framework for security applications. In *Applied Computer Security Associates ACSAC*, 2006.

[17] P. Li and S. Zdancewic. Practical Information-flow Control in Web-based Information Systems. In *Proceedings of 18th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2005.

[18] H. Mantel and A. Sabelfeld. A Unifying Approach to the Security of Distributed and Multi-Threaded Programs. *Journal of Computer Security*, 2002.

[19] R. Meushaw and D. Simard. Nettop - commercial technology in high assurance applications, 2000. `http://www.vmware.com/pdf/TechTrendNotes.pdf`.

[20] Sun Microsystems. Solaris trusted extensions. `http://www.sun.com`.

[21] Sun Microsystems. Trusted solaris operating environment - a technical overview. `http://www.sun.com`.

[22] A. C. Myers. Mostly-Static Decentralized Information Flow Control. Technical Report MIT/LCS/TR-783, Massachussetts Institute of Technology, January 1999. Ph.D. thesis.

[23] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java + information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[24] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[25] Security-enhanced Linux. `http://www.nsa.gov/selinux`.

[26] F. Pottier and V. Simonet. Information Flow Inference for ML. In *Proceedings ACM Symposium on Principles of Programming Languages*, pages 319–330, January 2002.

[27] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[28] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the IEEE Computer Security Foundations Workshop*, Aix-en-Provence, France, June 2005.

[29] G. Sally. Embedded Java with GCJ. *Linux Journal*, (145), May 2006.

[30] B. Sarna-Starosta and S.D. Stoller. Policy analysis for security-enhanced linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12, April 2004. Available at http://www.cs.sunysb.edu/~stoller/WITS2004.html.

[31] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Proceedings of the 2006 ISOC Networked and Distributed Systems Security Symposium (NDSS'06)*, San Diego, CA, USA, February 2006.

[32] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a linux security module. Technical Report 01-043, NAI Labs, 2001.

[33] G. Smith and D. Volpano. Secure Information Flow in a Multi-Threaded Imperative Language. In *Proceedings ACM Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, January 1998.

[34] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lapreau. The Flask architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*, pages 123–139, August 1999.

[35] Christopher Vance, Todd Miller, and Rob Dekelbaum. Security-Enhanced Darwin: Porting SELinux to Mac OS X. In *Proceedings of the Third Annual Security Enhanced Linux Symposium*, Baltimore, MD, USA, March 2007.

[36] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. 4(3):167–187, 1996.

[37] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using Replication and Partitioning to Build Secure Distributed Systems. In *Proceedings of the IEEE Symposium on Security and Privacy, 2003*, pages 236–250, 2003.

From STEM to SEAD: Speculative Execution for Automated Defense

Michael E. Locasto, Angelos Stavrou, Gabriela F. Cretu, Angelos D. Keromytis
Department of Computer Science, Columbia University
{locasto, angel, gcretu, angelos}@*cs.columbia.edu*

Abstract

Most computer defense systems crash the process that they protect as part of their response to an attack. Although recent research explores the feasibility of self-healing to automatically recover from an attack, self-healing faces some obstacles before it can protect legacy applications and COTS (Commercial Off–The–Shelf) software. Besides the practical issue of not modifying source code, self-healing must know both when to engage and how to guide a repair.

Previous work on a self-healing system, STEM, left these challenges as future work. This paper improves STEM's capabilities along three lines to provide practical speculative execution for automated defense (SEAD). First, STEM is now applicable to COTS software: it does not require source code, and it imposes a roughly 73% performance penalty on Apache's normal operation. Second, we introduce *repair policy* to assist the healing process and improve the semantic correctness of the repair. Finally, STEM can create behavior profiles based on aspects of data and control flow.

1 Introduction

Most software applications lack the ability to repair themselves during an attack, especially when attacks are delivered via previously unseen inputs or exploit previously unknown vulnerabilities. Self-healing software, an emerging area of research [31, 29, 28], involves the creation of systems capable of automatic remediation of faults and attacks. In addition to detecting and defeating an attack, self-healing systems seek to correct the integrity of the computation itself. Self-healing countermeasures serve as a first line of defense while a slower but potentially more complete human-driven response takes place.

Most self-healing mechanisms follow what we term the ROAR (Recognize, Orient, Adapt, Respond) workflow. These systems (*a*) *Recognize* a threat or attack has occurred, (*b*) *Orient* the system to this threat by analyzing it, (*c*) *Adapt* to the threat by constructing appropriate fixes or changes in state, and finally (*d*) *Respond* to the threat by verifying and deploying those adaptations.

While embryonic attempts in this space demonstrate the feasibility of the basic concept, these techniques face a few obstacles before they can be deployed to protect and repair legacy systems, production applications, and COTS (Commercial Off–The–Shelf) software. The key challenge is to apply a fix inline (*i.e.*, as the application experiences an attack) without restarting, recompiling, or replacing the process.

Executing through a fault in this fashion involves overcoming several obstacles. First, the system should not make changes to the application's source code. Instead, we supervise execution with dynamic binary rewriting. Second, the semantics of program execution must be maintained as closely as possible to the original intent of the application's author. We introduce *repair policy* to guide the semantics of the healing process. Third, the supervised system may communicate with external entities that are beyond the control or logical boundary of the self-healing system. We explore the design space of *virtual proxies* and detail one particular vector of implementation to address this problem. Finally, the system must employ detection mechanisms that can indicate when to supervise and heal the application's execution. Although STEM can operate with a number of detectors, we show how it gathers aspects of both data and control flow to produce an application's behavior profile.

1.1 Motivation

Our solutions are primarily motivated by the need to address the limitations of our previous self-healing software system prototype [31]. STEM (Selective Transactional EMulation) provides self-healing by speculatively executing "slices" of a process. We based this

first approach on a feedback loop that inserted calls to an x86 emulator at vulnerable points in an application's source code (requiring recompilation and redeployment). STEM supervises the application using microspeculation and error virtualization.

1.1.1 Microspeculation and Error Virtualization

The basic premise of our previous work is that portions of an application can be treated as a transaction. Functions serve as a convenient abstraction and fit the transaction role well in most situations [31]. Each transaction (vulnerable code slice) can be speculatively executed in a sandbox environment. In much the same way that a processor speculatively executes past a branch instruction and discards the mispredicted code path, STEM executes the transaction's instruction stream, optimistically "speculating" that the results of these computations are benign. If this *microspeculation* succeeds, then the computation simply carries on. If the transaction experiences a fault or exploited vulnerability, then the results are ignored or replaced according to the particular response strategy being employed. We call one such strategy, as discussed in previous work [31], *error virtualization*.

```
0  int bar(char* buf)
1  {
2    char rbuf[10];
3    int i=0;
4    if(NULL==buf)
5     return -1;
6    while(i<strlen(buf))
7    {
8     rbuf[i++]=*buf++;
9    }
10   return 0;
11 }
```

Figure 1: *Error Virtualization.* We can map unanticipated errors, like an exploit of the buffer overflow vulnerability in line 8, to anticipated error conditions explicitly handled by the existing program code (like the error condition return in line 5).

The key assumption underlying error virtualization is that a mapping can be created between the set of errors that *could* occur during a program's execution and the limited set of errors that the program code explicitly handles. By virtualizing errors, an application can continue execution through a fault or exploited vulnerability by nullifying its effects and using a manufactured return value for the function where the fault occurred. In the previous version of STEM, these return values were determined by source code analysis on the return type of the offending function. Vanilla error virtualization seems to work best with server applications — applications that typically have a request processing loop that can presumably tolerate minor errors in a particular trace of the loop. This paper, however, aims to provide a practical solution for client applications (*e.g.*, email, messaging, authoring, browsing) as well as servers.

1.1.2 Limitations of Previous Approach

Recently proposed approaches to self-healing such as *error virtualization* [31] and *failure-oblivious computing* [29] prevent exploits from succeeding by masking failures. However, error virtualization fails about 12% of the time, and both approaches have the potential for semantically incorrect execution. These shortcomings are devastating for applications that perform precise[1] calculations or provide authentication & authorization.

Furthermore, error virtualization required access to the source code of the application to determine appropriate error virtualization values and proper placement of the calls to the supervision environment. A better solution would operate on unmodified binaries and profile the application's behavior to learn appropriate error virtualization values during runtime.

Finally, as with all systems that rely on rewinding execution [4, 28] after a fault has been detected, I/O with external entities remains uncontrolled. For example, if a server program supervised by STEM writes a message to a network client during microspeculation, there is no way to "take back" the message: the state of the remote client has been irrevocably altered.

1.2 Contributions

The changes we propose and evaluate in this paper provide the basis for the redesign of STEM's core mechanisms as well as the addition of novel methods to guide the semantic correctness of the self-healing response. STEM essentially adds a policy-driven layer of indirection to an application's execution. The following contributions *collectively* provide a significant improvement over previous work:

1. **Eliminate Source-Level Modifications** – We employ error virtualization and microspeculation (and the new techniques proposed in this section) during binary rewriting. STEM serves as a self-contained environment for supervising applications without recompiling or changing source code.

2. **Virtual Proxies** – Self-healing techniques like microspeculation have difficulty "rewinding" the results of communication with remote entities that are not under the control of the self-healing system. This challenge can affect the semantic correctness of the healing process. We examine the notion of

virtual proxies to support cooperative microspeculation without changing the communications protocols or the code of the remote entity.

3. **Repair Policy** – Error virtualization alone is not appropriate for all functions and applications, especially if the function is not idempotent or if the application makes scientific or financial calculations or includes authentication & authorization checks. *Repair policy* provides a more complete approach to managing the semantic correctness of a repair. Section 4 describes the most relevant aspects and key features of repair policy as well as STEM's support for interpreting it. We refer the interested reader to our technical report [20] for a more complete discussion of the theoretical framework.

4. **Behavior Profiling** – Because we implement STEM in a binary supervision environment, we can non-invasively collect a profile of application behavior by observing aspects of both data and control flow. This profile assists in detection (detecting deviations from the profile), repair (selecting appropriate error virtualization values), and repair validation (making sure that future performance matches past behavior).

Using STEM to supervise dynamically linked applications directly from startup incurs a significant performance penalty (as shown in Table 2), especially for short-lived applications. Most of the work done during application startup simply loads and resolves libraries. This type of code is usually executed only once, and it probably does not require protection. Even though it may be acceptable to amortize the cost of startup over the lifetime of the application, we can work around the startup performance penalty by employing some combination of three reasonable measures: (1) statically linking applications, (2) only attaching STEM after the application has already started, (3) delay attaching until the system observes an IDS alert. We evaluate the second option by attaching STEM to Apache after Apache finishes loading. Our results (shown in Table 1) indicate that Apache experiences roughly a 73% performance degradation under STEM.

2 Related Work

Self-healing mechanisms complement approaches that stop attacks from succeeding by preventing the injection of code, transfer of control to injected code, or misuse of existing code. Approaches to automatically defending software systems have typically focused on ways to proactively protect an application from attack. Examples of these proactive approaches include writing the system in a "safe" language, linking the system with "safe" libraries [2], transforming the program with artificial diversity, or compiling the program with stack integrity checking [9]. Some defense systems also externalize their response by generating either vulnerability [8, 24, 10] or exploit [19, 22, 32, 36] signatures to prevent malicious input from reaching the protected system.

2.1 Protecting Control Flow

Starting with the technique of *program shepherding* [17], the idea of enforcing the integrity of control flow has been increasingly researched. Program shepherding validates branch instructions to prevent transfer of control to injected code and to make sure that calls into native libraries originate from valid sources. Control flow is often corrupted because input is eventually incorporated into part of an instruction's opcode, set as a jump target, or forms part of an argument to a sensitive system call. Recent work focuses on ways to prevent these attacks using tainted dataflow analysis [34, 25, 8].

Abadi *et al.* [1] propose formalizing the concept of Control Flow Integrity (CFI), observing that high-level programming often assumes properties of control flow that are not enforced at the machine level. CFI provides a way to statically verify that execution proceeds within a given control-flow graph (the CFG effectively serves as a policy). The use of CFI enables the efficient implementation of a software shadow call stack with strong protection guarantees. CFI complements our work in that it can enforce the invocation of STEM (rather than allowing malcode to skip past its invocation).

2.2 Self-Healing

Most defense mechanisms usually respond to an attack by terminating the attacked process. Even though it is considered "safe", this approach is unappealing because it leaves systems susceptible to the original fault upon restart and risks losing accumulated state.

Some first efforts at providing effective remediation strategies include failure oblivious computing [29], error virtualization [31], rollback of memory updates [32], crash-only software [5], and data structure repair [11]. The first two approaches may cause a semantically incorrect continuation of execution (although the Rx system [28] attempts to address this difficulty by exploring semantically safe alterations of the program's environment). Oplinger and Lam [26] employ hardware Thread-Level Speculation to improve software reliability. They execute an application's monitoring code in parallel with the primary computation and roll back the computation "transaction" depending on the results of the monitoring code. Rx employs proxies that are somewhat akin to our

virtual proxies, although Rx's are more powerful in that they explicitly deal with protocol syntax and semantics during replay.

ASSURE [30] is a novel attempt to minimize the likelihood of a semantically incorrect response to a fault or attack. ASSURE proposes the notion of *error virtualization rescue points*. A rescue point is a program location that is known to successfully propagate errors and recover execution. The insight is that a program will respond to malformed input differently than legal input; locations in the code that successfully handle these sorts of anticipated input "faults" are good candidates for recovering to a safe execution flow. ASSURE can be understood as a type of exception handling that dynamically identifies the best scope to handle an error.

2.3 Behavior-based Anomaly Detection

STEM also provides a mechanism to capture aspects of an application's behavior. This profile can be employed for three purposes: (a) to detect application misbehavior, (b) to aid self-healing, and (c) to validate the self-healing response and ensure that the application does not deviate further from its known behavior. STEM captures aspects of both control flow (via the execution context) and portions of the data flow (via function return values). This mechanism draws from a rich literature on host-based anomaly detection.

The seminal work of Hofmeyr, Somayaji, and Forrest [15, 33] examines an application's behavior at the system-call level. Most approaches to host-based intrusion detection perform anomaly detection [6, 13, 14] on sequences of system calls. The work of Feng *et al.* [12] includes an excellent overview of the literature circa 2003. The work of Bhatkar *et al.* [3] also contains a good overview of the more recent literature and offers a technique for *dataflow* anomaly detection to complement traditional approaches that concentrate mostly on control flow. Behavior profiling's logical goal is to create policies for detection [27, 18] and self-healing.

3 STEM

One of this paper's primary contributions is the reimplementation of STEM to make it applicable in situations where source code is not available. This section reviews the technical details of STEM's design and implementation. We built STEM as a tool for the IA-32 binary rewriting PIN [23] framework.

3.1 Core Design

PIN provides an API that exposes a number of ways to instrument a program during runtime, both statically (as a binary image is loaded) and dynamically (as each instruction, basic block, or procedure is executed). PIN tools contain two basic types of functions: (1) instrumentation functions and (2) analysis functions. When a PIN tool starts up, it registers instrumentation functions that serve as callbacks for when PIN recognizes an event or portion of program execution that the tool is interested in (*e.g.*, instruction execution, basic block entrance or exit, *etc.*). The instrumentation functions then employ the PIN API to insert calls to their analysis functions. Analysis functions are invoked every time the corresponding code slice is executed; instrumentation functions are executed only the first time that PIN encounters the code slice.

STEM treats each function as a transaction. Each "transaction" that should be supervised (according to policy) is speculatively executed. In order to do so, STEM uses PIN to instrument program execution at four points: function entry (*i.e.*, immediately before a CALL instruction), function exit (*i.e.*, between a LEAVE and RET instruction), immediately before the instruction *after* a RET executes, and for each instruction of a supervised function that writes to memory. The main idea is that STEM inserts instrumentation at both the start and end of each transaction to save state and check for errors, respectively. If microspeculation of the transaction encounters any errors (such as an attack or other fault), then the instrumentation at the end of the transaction invokes cleanup, repair, and repair validation mechanisms.

STEM primarily uses the "Routine" hooks provided by PIN. When PIN encounters a function that it has not yet instrumented, it invokes the callback instrumentation function that STEM registered. The instrumentation function injects calls to four analysis routines:

1. `STEM_Preamble()` – executed at the beginning of each function.
2. `STEM_Epilogue()` – executed before a RET instruction
3. `SuperviseInstruction()` – executed before each instruction of a supervised function
4. `RecordPreMemWrite()` – executed before each instruction of a supervised function that writes to memory

STEM's instrumentation function also intercepts some system calls to support the "CoSAK" supervision policy (discussed below) and the virtual proxies (discussed in Section 5).

3.2 Supervision Policy

One important implementation tradeoff is whether the decision to supervise a function is made at injection time

(*i.e.* during the instrumentation function) or at analysis time (*i.e.*, during an analysis routine). Consulting policy and making a decision in the latter (as the current implementation does) allows STEM to change the coverage supervision policy (that is, the set of functions it monitors) during runtime rather than needing to restart the application. Making the decision during injection time is possible, but not for all routines, and since the policy decision is made only once, the set of functions that STEM can instrument is not dynamically adjustable unless the application is restarted, or PIN removes all instrumentation and invokes instrumentation for each function again.

Therefore, each injected analysis routine determines dynamically if it should actually be supervising the current function. STEM instructs PIN to instrument *all* functions – a STEM analysis routine needs to gain control, even if just long enough to determine it should not supervise a particular function. The analysis routines invoke STEM's `ShouldSuperviseRoutine()` function to check the current supervision coverage policy in effect. Supervision coverage policies dictate which subset of an application's functions STEM should protect. These policies include:

- NONE – no function should be microspeculated
- ALL – all functions should be microspeculated
- RANDOM – a random subset should be microspeculated (the percentage is controlled by a configuration parameter)
- COSAK – all functions within a call stack depth of six from an input system call (*e.g.*, sys_read()) should be microspeculated[2]
- LIST – functions specified in a profile (either generated automatically by STEM or manually specified) should be microspeculated

In order to support the COSAK [16] coverage policy, STEM maintains a `cosak_depth` variable via four operations: check, reset, increment, and decrement. Every time an input system call is encountered, the variable is reset to zero. The variable is checked during `ShouldSuperviseRoutine()` if the coverage policy is set to COSAK. The variable is incremented every time a new routine is entered during `STEM_Preamble()` and decremented during `STEM_Epilogue()`.

3.3 STEM Workflow

Although STEM can supervise an application from startup, STEM benefits from using PIN because PIN can attach to a running application. For example, if a network sensor detects anomalous data aimed at a web server, STEM can attach to the web server process to protect it while that data is being processed. In this way, applications can avoid the startup costs involved in instrumenting shared library loading, and can also avoid the overhead of the policy check for most normal input.

STEM starts by reading its configuration file, attaching some command and control functionality (described in Section 3.4), and then registering a callback to instrument each new function that it encounters. STEM's basic algorithm is distributed over the four main analysis routines. If STEM operates in profiling mode (see Section 6), then these analysis routines remain unused.

3.3.1 Memory Log

Since STEM needs to treat each function as a transaction, undoing the effects of a speculated transaction requires that STEM keep a log of changes made to memory during the transaction. The memory log is maintained by three functions: one that records the "old" memory value, one that inserts a marker into the memory log, and one that rolls back the memory log and optionally restores the "old" values. STEM inserts a call to `RecordPreMemWrite()` before an instruction that writes to memory. PIN determines the size of the write, so this analysis function can save the appropriate amount of data. Memory writes are only recorded for functions that should be supervised according to coverage policy. During `STEM_Preamble()`, PIN inserts a call to `InsertMemLogMarker()` to delimit a new function instance. This marker indicates that the last memory log maintenance function, `UnrollMemoryLog()`, should stop rollback after it encounters the marker. The rollback function deletes the entries in the memory log to make efficient use of the process's memory space. This function can also restore the "old" values stored in the memory log in preparation for repair.

3.3.2 STEM_Preamble()

This analysis routine performs basic record keeping. It increments the COSAK depth variable and maintains other statistics (number of routines supervised, *etc.*). Its most important tasks are to (1) check if supervision coverage policy should be reloaded and (2) insert a function name marker into the memory log if the current function should be supervised.

3.3.3 STEM_Epilogue()

STEM invokes this analysis routine immediately before a return instruction. Besides doing its part to maintain the COSAK depth variable, this analysis routine ensures that

the application has a chance to self-heal before a transaction is completed. If the current function is being supervised, this routine interprets the application's repair policy (a form of integrity policy based on extensions to the Clark-Wilson integrity model, see Section 4 for details), invokes the repair procedure, and invokes the repair validation procedure. If both of these latter steps are successful or no repair is needed, then the transaction is considered to be successfully committed. If not, and an error *has* occurred, then STEM falls back to crashing the process (the current state of the art) by calling `abort()`.

This analysis routine delegates the setup of error virtualization to the repair procedure. The repair procedure takes the function name, current architectural context (*i.e.*, CPU register values), and a flag as input. The flag serves as an indication to the repair procedure to choose between normal cleanup or a "self-healing" cleanup. While normal cleanup always proceeds from `STEM_Epilogue()`, a self-healing cleanup can be invoked synchronously from `STEM_Epilogue()` or asynchronously from a signal handler. The latter case usually occurs when STEM employs a detector that causes a signal such as SIGSEGV to occur when it senses an attack.

Normal cleanup simply entails deleting the entries for the current function from the memory log. If self-healing is needed, then the values from the memory log are restored. In addition, a flag is set indicating that the process should undergo error virtualization, and the current function name is recorded.

3.3.4 SuperviseInstruction()

The job of this analysis routine is to intercept the instruction that immediately follows a RET instruction. By doing so, STEM allows the RET instruction to operate as it needs to on the architectural state (and by extension, the process stack). After RET has been invoked, if the flag for error virtualization is set, then STEM looks up the appropriate error virtualization value according to policy (either a vanilla EV value, or an EV value derived from the application's profile or repair policy). STEM then performs error virtualization by adjusting the value of the `%eax` register and resets the error virtualization flag. STEM ensures that the function returns appropriately by comparing the return address with the saved value of the instruction pointer immediately after the corresponding CALL instruction.

3.4 Additional Controls

STEM includes a variety of control functionality that assists the core analysis routines. The most important of these additional components intercepts signals to deal with dynamically loading configuration and selecting a suitable error virtualization value.

STEM defines three signal handlers and registers them with PIN. The first intercepts SIGUSR1 and sets a flag indicating that policy and configuration should be reloaded, although the actual reload takes place during the execution of the next `STEM_Preamble()`. The second signal handler intercepts SIGUSR2 and prints some runtime debugging information. The third intercepts SIGSEGV (for cases where detectors alert on memory errors, such as address space randomization). The handler then causes the repair procedure to be invoked, after it has optionally asked the user to select a response as detailed by the repair policy. Part of the response can include forwarding a snapshot of memory state to support automatically generating an exploit signature as done with the previous version of STEM for the FLIPS system [22].

STEM supports a variety of detection mechanisms, and it uses them to measure the integrity of the computation at various points in program execution and set a flag that indicates `STEM_Epilogue()` should initiate a self-healing response. Our current set of detectors includes one that detects an anomalous set of function calls (*i.e.*, a set of functions that deviate from a profile learned when STEM is in profiling mode) as well as a shadow stack that detects integrity violations of the return address or other stack frame information. STEM also intercepts a SIGSEGV produced by an underlying OS that employs address space randomization. We are currently implementing tainted dataflow analysis. This detector requires more extensive instrumentation, thereby limiting the supervision coverage policy to "ALL."

4 Repair Policy

Achieving a semantically correct response remains a key problem for self-healing systems. Executing through a fault or attack involves a certain amount of risk. Even if software could somehow ignore the attack itself, the best sequence of actions leading back to a safe state is an open question. The exploit may have caused a number of changes in state that corrupt execution integrity before an alert is issued. Attempts to self-heal must not only stop an exploit from succeeding or a fault from manifesting, but also repair execution integrity as much as possible. However, self-healing strategies that execute through a fault by effectively pretending it can be handled by the program code or other instrumentation may give rise to semantically incorrect responses. In effect, naive self-healing may provide a cure worse than the disease.

Figure 2 illustrates a specific example: an error may exist in a routine that determines the access control rights for a client. If this fault is exploited, a self-healing tech-

```
int login(UCRED creds)
{
  int authenticated = check_credentials(creds);
  if(authenticated) return login_continue();
  else return login_reject();
}
int check_credentials(UCRED credentials)
{
  strcpy(uname, credentials.username);
  return checkpassword(lookup(uname), credentials);
}
```

Figure 2: *Semantically Incorrect Response.* If an error arising from a vulnerability in `check_credentials` occurs, a self-healing mechanism may attempt to return a simulated error code from `check_credentials`. Any value other than 0 that gets stored in `authenticated` causes a successful login. What may have been a simple DoS vulnerability has been transformed into a valid login session by virtue of the "security" measures. STEM interprets *repair policy* to intelligently constrain return values and other application data.

nique like error virtualization may return a value that allows the authentication check to succeed. This situation occurs precisely because the recovery mechanism is oblivious to the semantics of the code it protects.

One solution to this problem relies on annotating the source code to (a) indicate which routines should not be "healed" or (b) to provide appropriate return values for such sensitive functions, but we find these techniques unappealing because of the need to modify source code. Since source-level annotations serve as a vestigial policy, we articulate a way to augment self-healing approaches with the notion of *repair policy*. A repair policy (or a recovery policy – we use the terms interchangeably) is specified separately from the source code and describes how execution integrity should be maintained after an attack is detected. Repair policy can provide a way for a user to customize an application's response to an intrusion attempt and can help achieve a completely automated recovery.

4.1 Integrity Repair Model

We provide a theoretical framework for repair policy by extending the Clark-Wilson Integrity Model (CW) [7] to include the concepts of (a) repair and (b) repair validation. CW is ideally suited to the problem of detecting when constraints on a system's behavior and information structures have been violated. The CW model defines rules that govern three major constructs: constrained data items (CDI), transformation procedures (TP), and integrity verification procedures (IVP). An information system is composed of a set of TPs that transition CDIs from one valid state to another. The system also includes IVPs that measure the integrity of the CDIs at various points of execution.

Although a TP should move the system from one valid state to the next, it may fail for a number of reasons (incorrect specification, a vulnerability, hardware faults, *etc.*). The purpose of an IVP is to detect and record this failure. CW does not address the task of returning the system to a valid state or formalize procedures that *restore* integrity. In contrast, repair policy focuses on ways to recover after an unauthorized modification. Our extensions supplements the CW model with primitives and rules for recovering from a policy violation and validating that the recovery was successful.

4.2 Interpreting Repair Policy

STEM interprets repair policy to provide a mechanism that can be selectively enforced and retrofitted to the protected application without modifying its source code (although *mapping* constraints to source-level objects assists in maintaining application semantics). As with most self-healing systems, we expect the repairs offered by this "behavior firewall" to be temporary constraints on program behavior — emergency fixes that await a more comprehensive patch from the vendor. One advantage of repair policy is that an administrator can "turn off" a broken repair policy without affecting the execution of the program — unlike a patch.

Repair policy is specified in a file external to the source code of the protected application and is used only by STEM (*i.e.*, the compiler, the linker, and the OS are not involved). This file describes the legal settings for variables in an aborted transaction. The basis of the policy is a list of relations between a transaction and the CDIs that need to be adjusted after error-virtualization, including the return address and return value. A complete repair policy is a wide-ranging topic; in this paper we consider a simple form that:

1. specifies appropriate error virtualization settings to avoid an incorrect return value that causes problems like the one illustrated in Figure 2
2. provides memory rollback for an aborted transaction
3. sets memory locations to particular values

Figure 3 shows a sample policy for our running example. The first statement defines a symbolic value. The latter three statements define an IVP, RP, and TP. The IVP defines a simple detector that utilizes STEM's shadow stack. The RP sets the return value to a semantically correct value and indicates that memory changes should be undone, and the TP definition links these measurement and repair activities together. An RP can contain a list of

```
symval AUTHENTICATION_FAILURE = 0;
ivp MeasureStack :=:
  ('raddress==*shadowstack[0]);
rp FixAuth :=:
  ('rvalue==AUTHENTICATION_FAILURE),
  (rollback);
tp check_credentials
  &MeasureStack :=: &FixAuth;
```

Figure 3: *Sample Repair Policy.* If the TP named `check_credentials` fails, then the memory changes made during this routine are reset and STEM stores the value 0 in the return value (and thus into `authenticated`), causing the login attempt to fail.

asserted conditions on CDIs that should be true after self-healing completes. The example illustrates the use of the special variable `'rvalue` (the apostrophe distinguishes it from any CDI named `rvalue`). This variable helps customize vanilla error virtualization to avoid problems similar to the one in Figure 2.

4.3 Limitations and Future Work

Our future work on STEM centers on improving the power and ease of use of repair policy. We intend to provide a mapping between memory layout and source-level variables. Cutting across layers of abstraction like this requires augmenting the mapping mechanism with a type system and the ability to handle variables that do not reside at fixed addresses. Second, while virtual proxies are a key aid to provide a semantically correct response, there is no explicit integration of virtual proxy behavior with repair policy specification. Third, we intend to explore the addition of formal logic to STEM so that it can reason about the constraints on the data involved in a transaction to learn the best response over time.

Finally, the information that a particular set of variables have been corrupted raises the possibility of notifying other hosts and application instances to proactively invoke repair procedures in order to protect against a widespread attack [21, 8, 35]. This sort of detection is helpful in creating a system that automatically tunes the security posture of an organization.

5 Virtual Proxies

Attempts to sandbox an application's execution must sooner or later allow the application to deal with global input and output sources and sinks that are beyond the control of the sandbox. Microspeculation becomes unsafe when the speculated process slice communicates with entities beyond the control of STEM. If a transaction is not idempotent (*i.e.*, it alters global state such as shared memory, network messages, *etc.*), then microspeculation must stop before that global state is changed. The self-healing system can no longer safely speculate a code slice: the results of execution up to that point must be committed, thus limiting microspeculation's effective scope.

Repair attempts may fall short in situations where an exploit on a machine (*e.g.*, an electronic funds transfer front-end) that is being "healed" has visible effects on another machine (*e.g.*, a database that clears the actual transfer). For example, if a browser exploit initiates a PayPal transaction, even though STEM can recover control on the local machine, the user will not have an automated recourse with the PayPal system.

Such situations require additional coordination between the two systems – microspeculation must span both machines. If both machines reside in the same administrative domain, achieving this *cooperative microspeculation* is somewhat easier, but we prefer a solution that works for situations like the PayPal example. While a self-healing system can record I/O, it cannot ask a communications partner to replay input or re-accept output. Doing so requires that the protocol (and potentially the network infrastructure) support speculative messaging and entails changing the partner's implementation so that it can rewind its own execution. Since STEM may not be widely deployed, we cannot rely on this type of explicit cooperation.

5.1 Solutions

We can achieve cooperative microspeculation in at least four ways, each of which expresses a tradeoff between semantic correctness and invasiveness.

1. **Protocol Modification** – Modify network or filesystem protocols and the network infrastructure to incorporate an explicit notion of speculation.

2. **Modify Communications Partner** – Modify the code of the remote entity so that it can cooperate when the protected application is microspeculating, and thus anticipate when it may be sending or receiving a "speculated" answer or request.

3. **Gradual Commits** – Transactions can be continuously limited in scope. All memory changes occurring *before* an I/O call are marked as not undoable. Should the microspeculated slice fail, STEM only undoes changes to memory made after the I/O call.

4. **Virtual Proxies** – Use buffers to record and replay I/O locally. Virtual proxies effectively serve as a man-in-the-middle during microspeculation to delay the effects of I/O on the external world.

While some network and application-level protocols may already include a notion of "replay" or speculative execution, implementing widespread changes to protocol specifications and the network infrastructure is fairly invasive. Nevertheless, it presents an interesting technical research challenge. Another interesting possibility is to modify the execution environment or code of the remote communications partner to accept notifications from a STEM-protected application. After receiving the notification, the remote entity speculates its own I/O. While this approach promises a sound solution, it violates our transparency requirements.

We choose to use a combination of virtual proxies and gradual commits because these solutions have the least impact on current application semantics and require a straightforward implementation. Since we are already "modifying" the local entity, we can avoid modifying the remote entity or any protocols. Using gradual commits and virtual proxies constrains the power of our solution, but we believe it is an acceptable tradeoff, especially as self-healing systems gain traction – they should perturb legacy setups as little as possible.

5.2 Design

I/O system calls that occur during the speculated portion of a process constitute a challenge for safely discarding speculated operations should an exploit occur. While speculation can immediately resume after an I/O call, the I/O call itself cannot be replayed or undone. If a fault or exploit occurs after the I/O call (but still in the microspeculated routine), then STEM cannot rewind to the beginning of the code slice. Rather, it can only unwind back to the I/O call. Memory and other state changes before the I/O call must remain in effect (we ignore for the moment explicit changes made as part of repair policy). This gradual process of commits is one way in which we can attempt to control uncertainty in the correctness of the response.

A virtual proxy serves as a delegate for a communications partner (*e.g.*, server, client, or peer) for the program that STEM is supervising. A virtual proxy is composed of a set of functions that modify a buffer that is bound during the scope of a supervised routine. The primary function of the virtual proxy is to allow STEM, as it speculates a slice of an application, to "take back" some output or "push back" some input. As a proof of concept, our current implementation only intercepts `read` and `write` calls. Virtual proxies are designed to handle this two-part problem.

Virtual Proxy Input In this case, an external component (such as a filesystem) is providing input. The code slice that contains this input call can either (a) successfully complete without an error or exploit, or (b) experience such a fault and have STEM attempt repair. In case (a), nothing need happen because STEM's state is consistent with the global state. In case (b), STEM must attempt a semantically correct repair – regardless of whether or not the input was legal or malformed/malicious. At this point, the external entity believes its state has changed (and therefore will not replay the input). In the optimal case, STEM should continue executing with what input *that was supposed to be consumed by the transaction removed from the input buffer.* Naturally, STEM cannot determine this on its own (and the speculated code slice is no help either – it evidently experienced a fault when processing this input). Instead, STEM can continue processing and draw from the virtual proxy's buffers during the next input request.

Virtual Proxy Output In order to deal with speculated output, STEM must buffer output until it requires input from the external component. At this point, STEM must allow the remote partner to make progress. This process of gradual commits is useful, but has the potential to delay too long and cause an application-level timeout. STEM does not currently deal with this issue. As with virtual proxy input, the speculated slice can (a) successfully complete without an error or exploit, or (b) experience such a fault and have STEM attempt a repair. In case (a), gradual commits suffice, as the output calls simply finish. In case (b), the external component has been given a message it should not have. If the virtual proxy were not operating, a STEM-supervised application would need to ask for that output to be ignored. The virtual proxy allows STEM to buffer output until the microspeculated slice successfully completes. If the slice fails, then STEM instructs the virtual proxy to discard the output (or replace it).

5.3 Limitations and Future Work

Although virtual proxies help address the external I/O problem for microspeculation, they are not a perfect solution. In the case where STEM is supervising the processing of input, the virtual proxy can only buffer a limited amount of input – and it is not clear how to selectively discard portions of that input should a transaction fail. In the cases where STEM supervises the sending of output, the virtual proxy buffers the output until STEM requests input from the remote communications partner. At this point, STEM has reached the edge of our ability to safely microspeculate, and without further support in the virtual proxy that explicitly communicates with the remote partner, STEM must stop speculating and finally give the data to the remote partner.

One interesting problem is to use multiple virtual proxies to classify and identify multiple conversation streams. This information is not present at the level of

read and write system calls, and STEM would need to break through layers of abstraction to support this ability. Finally, since the virtual proxy is under STEM's control, STEM can attempt to construct a memory and behavior model of the remote communications partner to determine if it is behaving in a malicious fashion.

6 Behavior Models

Although STEM uses a number of detection strategies (including a shadow stack), STEM also provides for host-based anomaly detection. This type of detection helps identify previously unknown vulnerabilities and exploits, but depends on the system having a model or profile of normal behavior. STEM collects aspects of data and control flow to learn an application's behavior profile. STEM can leverage the information in the profile to detect misbehavior (*i.e.,* deviation from the profile) and automatically validate repairs to ensure that self-healing achieves normal application behavior.

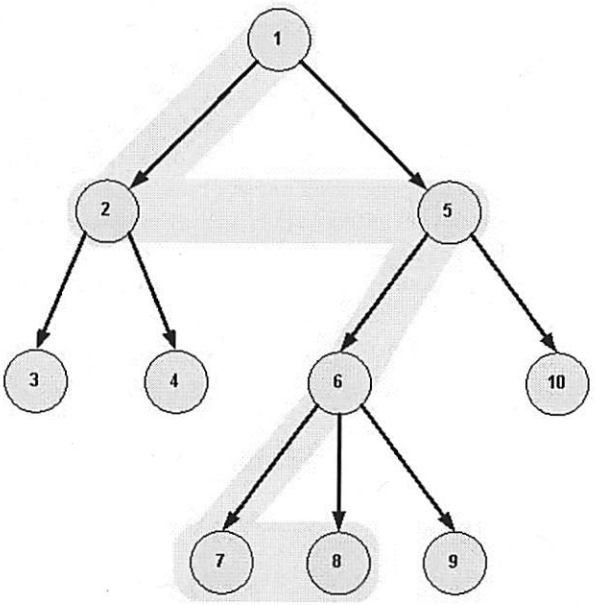


Figure 4: *Example of Computing Execution Window Context.* Starting from function 8, we traverse the graph beginning from the previously executed siblings up to the parent. We recursively repeat this algorithm for the parent until we either reach the window width or the root. In this example, the window contains functions 7, 6, 5, 2, 1. Systems that examine the call stack would only consider 6, 5, and 1 at this point.

In profiling mode, STEM dynamically analyzes all function calls made by the process, including regular functions and library calls as well as system calls. Previous work typically examines only system calls or is driven by static analysis. STEM collects a feature set that includes a mixture of parent functions and previous sibling functions. STEM generates a record of the observed return values for various invocations of each function.

A behavior profile is a graph of execution history records. Each record contains four data items: an identifier, a return value, a set of argument values, and a context. Each function name serves as an identifier (although address/callsites can also be used). A mixture of parents and previous siblings compose the context. The argument and return values correspond to the argument values at the time that function instance begins and ends, respectively. STEM uses a pair of analysis functions (inserted at the start and end of each routine) to collect the argument values, the function name, the return value, and the function context.

Each record in the profile helps to identify an instance of a function. The feature set "unflattens" the function namespace of an application. For example, `printf()` appears many times with many different contexts and return values, making it hard to characterize. Considering every occurrence of `printf()` to be the same instance reduces our ability to make predictions about its behavior. On the other hand, considering all occurrences of `printf()` to be separate instances combinatorially increases the space of possible behaviors and similarly reduces our ability to make predictions about its behavior in a reasonable amount of time. Therefore, we need to construct an "execution context" for each function based on both control (predecessor function calls) and data (return & argument values) flow. This context helps collapse *occurrences* of a function into an *instance* of a function. Figure 4 shows an example context window.

During training, one behavior aspect that STEM learns is which return values to predict based on execution contexts of varying window sizes. The general procedure attempts to compute the prediction score by iteratively increasing the window size and seeing if additional information is revealed by considering the extra context.

We define the return value "predictability score" as a value from zero to one. For each context window, we calculate the "individual score": the relative frequency of this particular window when compared with the rest of the windows leading to a function. The predictability score for a function F is the sum of the individual scores that lead to a single return value. Figure 5 displays an example of this procedure. We do not consider windows that contain smaller windows leading to a single return value since the information that they impart is already subsumed by the smaller execution context. For example, in Figure 5, we do not consider all windows with a suffix of AF (*i.e.,* $*AF$).

Limitations STEM relies on PIN to reliably detect

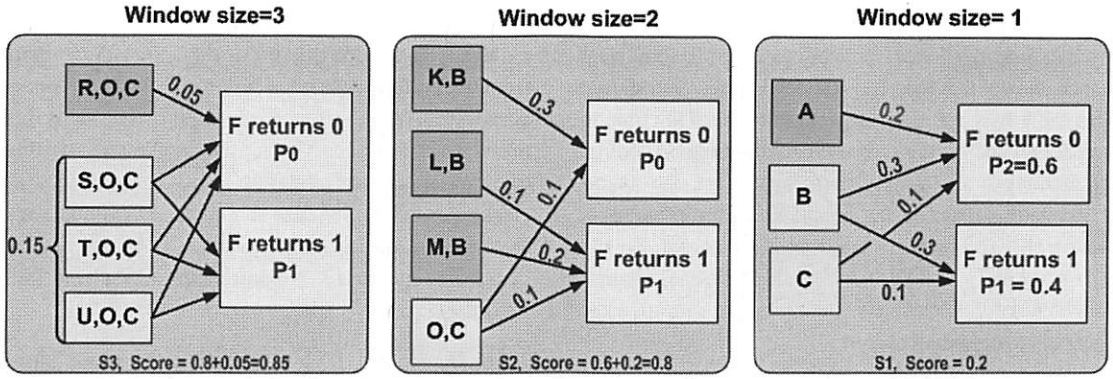


Figure 5: *Example of Computing Return Value Predictability (predictability score).* The figure illustrates the procedure for function F and for two return values 0 & 1 for three window sizes. The arrow labels indicate what percentage of instances for the given window will lead to the return value of F when compared with the rest of the windows. For window size 1 (S1) we have three predicate functions (A, B, and C) with only one, A, leading to a unique return value with score 0.2. This score is the relative frequency of window AF,[2] when compared with all other windows leading to F, for all return values. We add a score to the total score when a window leads to single return value of F since this situation is the only case that "predicts" a return value. We consider only the smallest windows that lead to a single value (*e.g.*, A is no longer considered for S2 and KB, LB, MB for S3) because larger windows do not add anything to our knowledge for the return value.

returns from a function. Detecting function exit is difficult in the presence of optimizations like tail recursion. Also, since the generated profile is highly binary-dependent, STEM should recognize when an older profile is no longer applicable (and a new one needs to be built), *e.g.*, as a result of a new version of the application being rolled out, or due to the application of a patch.

7 Evaluation

The goal of our evaluation is to characterize STEM's impact on the normal performance of an application. STEM incurs a relatively low performance impact for real-world software applications, including both interactive desktop software as well as server programs. Although the time it takes to self-heal is also of interest, our experiments on synthetic vulnerabilities show that this amount of time depends on the complexity of the repair policy (*i.e.*, how many memory locations need to be adjusted) and the memory log rollback. Even though memory log rollback is an $O(n)$ operation (we discuss a possible optimization below), STEM's self-healing and repair procedure usually takes under a second (using the `x86 rdtsc` instruction we observe an average of 15 milliseconds) to interpret the repair policy for these vulnerabilities.

Of more general concern is whether or not STEM slows an application down to the point where it becomes apparent to the end-user. Even though STEM has a rather significant impact on an application's startup time (as shown in Table 2), STEM does not have a human-discernible impact when applied to regular application

Table 1: *Impact on Apache Excluding Startup.* We tested STEM's impact on two versions of Apache by starting Apache in single-threaded mode (to force all requests to be serviced sequentially by the same thread). We then attach STEM after verifying that Apache has started by viewing the default homepage. We use `wget` to recursively retrieve the pages of the online manual included with Apache. The total downloaded material is roughly 72 MB in about 4100 files. STEM causes a 74.85% slowdown, far less than the tens of thousands factor when including startup. Native execution of Apache 2.0.53 takes 0.0626 seconds per request; execution of the same under STEM takes 0.1095 seconds per request. For a newer version of Apache (2.2.4), we observe a slight improvement to 72.54%.

Apache	Native (s)	STEM (s)	Impact %
`v2.0.53`	3746	6550	74.85%
`v2.2.4`	16215	27978	72.54%

operations. For example, Firefox remains usable for casual web surfing when operating with STEM. In addition, playing a music file with `aplay` also shows no sign of sound degradation – the only noticeable impact comes during startup. Disregarding this extra time, the difference between `aplay`'s native performance and its performance under STEM is about 2 seconds. If STEM is attached to `aplay` after the file starts playing, there is an eight second delay followed by playback that proceeds with only a 3.9% slowdown. Most of the performance penalty shown in Table 2 and Table 3 is exaggerated by

Table 2: *Performance Impact Data.* Attaching STEM at startup to dynamically linked applications incurs a significant performance penalty that lengthens the total application startup time. This table lists a variety of programs where the time to execute is dominated by the increased startup time. Although most applications suffer a hefty performance hit, the majority of the penalty occurs during application startup and exit. Note that `aplay` shows fairly good performance; a roughly six-minute song plays in STEM for 88 seconds longer than it should – with 86 of those seconds coming during startup, when the file is not actually being played.

Application	Native (s)	STEM (s)	Slowdown
aplay	371.0125	459.759	0.239
arch	0.001463	14.137	9662.021
xterm	0.304	215.643	708.352
echo	0.002423	17.633	7276.342
false	0.001563	16.371	10473.088
Firefox	2.53725	70.140	26.644
gzip-h	4.51	479.202	105.253
gzip-k	0.429	58.954	136.422
gzip-d	2.281	111.429	47.851
md5-k	0.0117	32.451	2772.589
md5-d	0.0345	54.125	1567.841
md5-h	0.0478	70.883	1481.908
ps	0.0237	44.829	1890.519
true	0.001552	16.025	10324.387
uname	0.001916	19.697	10279.271
uptime	0.002830	27.262	9632.215
date	0.001749	26.47	15133.362
id	0.002313	24.008	10378.592

Table 3: *Performance Without (Some) Startup.* We remove a well-defined portion of the application's initialization from the performance consideration in Table 2. Removing supervision of this portion of the startup code improves performance over full supervision. The remaining run time is due to a varying amount of startup code, the application itself, and cleanup/exit code. In order to completely eliminate application startup from consideration, we attach to Apache after its initialization has completed. We present those results in Table 1.

Application	STEM-init (s)	Revised Slowdown
arch	3.137	2143.22
xterm	194.643	639.273
echo	5.633	2323.803
false	4.371	2795.545
Firefox	56.14	21.128
gzip-h	468.202	102.814
gzip-k	47.954	110.780
gzip-d	100.429	43.025
md5-k	20.451	1746.948
md5-d	42.125	1220.014
md5-h	58.883	1230.862
ps	31.829	1341.996
true	5.025	3236.758
uname	8.697	4538.144
uptime	15.262	5391.932
date	14.47	8272.299
id	13.008	5622.865

the simple nature of the applications. Longer-running applications experience a much smaller impact relative to total execution, as seen by the gzip, md5sum, and Firefox results.

Most of the work done during startup loads and resolves libraries for dynamically linked applications. STEM can avoid instrumenting this work (and thus noticeably reduce startup time) in at least two ways. The first is to simply not make the application dynamically linked. We observed for some small test applications (including a program that incorporates the example shown in Figure 2 from Section 4) that compiling them as static binaries reduces execution time from fifteen seconds to about five seconds. Second, since PIN can attach to applications after they have started (in much the same way that a debugger does), we can wait until this work completes and then attach STEM to protect the mainline code execution paths. We used this capability to attach STEM to Firefox and Apache after they finish loading (we measured the performance impact on Apache using this method; see Table 1). Also, as mentioned in Section 3, we can allow the application to begin executing normally and only attach STEM when a network anomaly detector issues an IDS alert. Finally, it may be acceptable for certain long-running applications (*e.g.*, web, mail, database, and DNS servers) to amortize this long startup time (on the order of minutes) over the total execution time (on the order of weeks or months).

7.1 Experimental Setup

We used multiple runs of applications that are representative of the software that exists on current Unix desktop environments. We tested aplay, Firefox, gzip, md5sum, and xterm, along with a number of smaller utilities: arch, date, echo, false, true, ps, uname, uptime, and id. The applications were run on a Pentium M 1.7 GHz machine with 2 GB of memory running Fedora Core 3 Linux. We used a six minute and ten second WAV file to test `aplay`. To test both `md5sum` and `gzip`, we used three files: *httpd-2.0.53.tar.gz*, a Fedora Core kernel (*vmlinuz-2.6.10-1.770_FC3smp*), and

the */usr/share/dict/linux.words* dictionary. Our Firefox instance simply opened a blank page. Our xterm test creates an xterm and executes the `exit` command. We also tested two versions of `httpd` (2.0.53 and 2.2.4) by attaching STEM after Apache starts and using `wget` to recursively download the included manual from another machine on the same network switch. Doing so gives us a way to measure STEM's impact on normal performance excluding startup (shown in Table 1). In the tables, the suffixes for gzip and md5sum indicate the kernel image (k), the httpd tarball (h), and the dictionary (d).

Memory Log Enhancements We can improve performance of supervised routines by modifying the memory log implementation (currently based on a linked list). One way to improve performance is to preallocate memory slots based on the typical memory use of each supervised function. If we can bound the number of stores in a piece of code (*e.g.*, because STEM or another profiling tool has observed its execution), then STEM can preallocate an appropriately sized buffer.

8 Conclusion

Self-healing systems face a number of challenges before they can be applied to legacy applications and COTS software. Our efforts to improve STEM focus on four specific problems: (1) applying STEM's microspeculation and error virtualization capabilities in situations where source code is unavailable, (2) helping create a behavior profile for detection and repair, (3) improving the correctness of the response by providing a mechanism to interpret *repair policy*, and (4) implementing *virtual proxies* to help deal with speculated I/O. These solutions collectively provide a more streamlined version of STEM that represents a significant improvement in both features and performance: our current implementation imposes a 74% impact for whole-application supervision (versus the previous 30% impact for a single supervised routine and a 3000X slowdown for whole-application supervision).

Acknowledgments

We deeply appreciate the insightful and constructive comments made by the anonymous reviewers. This material is based on research sponsored by the Air Force Research Laboratory under agreement number FA8750-06-2-0221, and by the National Science Foundation under NSF grants CNS-06-27473, CNS-04-26623 and CCF-05-41093. We authorize the U.S. Government to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2005).

[2] BARATLOO, A., SINGH, N., AND TSAI, T. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference* (June 2000).

[3] BHATKAR, S., CHATURVEDI, A., AND SEKAR., R. Improving Attack Detection in Host-Based IDS by Learning Properties of System Call Arguments. In *Proceedings of the IEEE Symposium on Security and Privacy* (2006).

[4] BROWN, A., AND PATTERSON, D. A. Rewind, Repair, Replay: Three R's to dependability. In 10th *ACM SIGOPS European Workshop* (Saint-Emilion, France, Sept. 2002).

[5] CANDEA, G., AND FOX, A. Crash-Only Software. In *Proceedings of the* 9th *Workshop on Hot Topics in Operating Systems (HOTOS-IX)* (May 2003).

[6] CHARI, S. N., AND CHENG, P.-C. BlueBoX: A Policy–driven, Host–Based Intrusion Detection System. In *Proceedings of the* 9th *Symposium on Network and Distributed Systems Security (NDSS 2002)* (2002).

[7] CLARK, D. D., AND WILSON, D. R. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the IEEE Symposium on Security and Privacy* (1987).

[8] COSTA, M., CROWCROFT, J., CASTRO, M., AND ROWSTRON, A. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)* (2005).

[9] COWAN, C., PU, C., MAIER, D., HINTON, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the USENIX Security Symposium* (1998).

[10] CUI, W., PEINADO, M., WANG, H. J., AND LOCASTO, M. E. ShieldGen: Automated Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2007).

[11] DEMSKY, B., AND RINARD, M. C. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the* 18th *Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications* (October 2003).

[12] FENG, H. H., KOLESNIKOV, O., FOGLA, P., LEE, W., AND GONG, W. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (May 2003).

[13] GAO, D., REITER, M. K., AND SONG, D. Gray-Box Extraction of Execution Graphs for Anomaly Detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2004).

[14] GIFFIN, J. T., DAGON, D., JHA, S., LEE, W., AND MILLER, B. P. Environment-Sensitive Intrusion Detection. In *Proceedings of the* 8th *International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2005).

[15] HOFMEYR, S. A., SOMAYAJI, A., AND FORREST, S. Intrusion Detection System Using Sequences of System Calls. *Journal of Computer Security 6*, 3 (1998), 151–180.

[16] HTTP://SERG.CS.DREXEL.EDU/COSAK/INDEX.SHTML.

[17] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure Execution Via Program Shepherding. In *Proceedings of the* 11th *USENIX Security Symposium* (August 2002).

[18] LAM, L. C., AND CKER CHIUEH, T. Automatic Extraction of Accurate Application-Specific Sandboxing Policy. In *Proceedings of the* 7th *International Symposium on Recent Advances in Intrusion Detection* (September 2004).

[19] LIANG, Z., AND SEKAR, R. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In *Proceedings of the* 12th *ACM Conference on Computer and Communications Security (CCS)* (November 2005).

[20] LOCASTO, M. E., CRETU, G. F., STAVROU, A., AND KEROMYTIS, A. D. A Model for Automatically Repairing Execution Integrity. Tech. Rep. CUCS-005-07, Columbia University, January 2007.

[21] LOCASTO, M. E., SIDIROGLOU, S., AND KEROMYTIS, A. D. Software Self-Healing Using Collaborative Application Communities. In *Proceedings of the* 13th *Symposium on Network and Distributed System Security (NDSS 2006)* (February 2006), pp. 95–106.

[22] LOCASTO, M. E., WANG, K., KEROMYTIS, A. D., AND STOLFO, S. J. FLIPS: Hybrid Adaptive Intrusion Prevention. In *Proceedings of the* 8th *International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2005), pp. 82–101.

[23] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD., K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of Programming Language Design and Implementation (PLDI)* (June 2005).

[24] NEWSOME, J., BRUMLEY, D., AND SONG, D. Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the* 13th *Symposium on Network and Distributed System Security (NDSS 2006)* (February 2006).

[25] NEWSOME, J., AND SONG, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the* 12th *Symposium on Network and Distributed System Security (NDSS)* (February 2005).

[26] OPLINGER, J., AND LAM, M. S. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the* 10th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)* (October 2002).

[27] PROVOS, N. Improving Host Security with System Call Policies. In *Proceedings of the* 12th *USENIX Security Symposium* (August 2003), pp. 207–225.

[28] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)* (2005).

[29] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D., LEU, T., AND W BEEBEE, J. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings* 6th *Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004).

[30] SIDIROGLOU, S., LAADAN, O., KEROMYTIS, A. D., AND NIEH, J. Using Rescue Points to Navigate Software Recovery (Short Paper). In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2007).

[31] SIDIROGLOU, S., LOCASTO, M. E., BOYD, S. W., AND KEROMYTIS, A. D. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference* (April 2005), pp. 149–161.

[32] SMIRNOV, A., AND CHIUEH, T. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *Proceedings of the* 12th *Symposium on Network and Distributed System Security (NDSS)* (February 2005).

[33] SOMAYAJI, A., AND FORREST, S. Automated Response Using System-Call Delays. In *Proceedings of the* 9th *USENIX Security Symposium* (August 2000).

[34] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the* 11th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)* (October 2004).

[35] TUCEK, J., LU, S., HUANG, C., XANTHOS, S., ZHOU, Y., NEWSOME, J., BRUMLEY, D., AND SONG, D. Sweeper: A Lightweight End-to-End System for Defending Against Fast Worms. In *EuroSys* (2007).

[36] XU, J., NING, P., KIL, C., ZHAI, Y., AND BOOKHOLT, C. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In *Proceedings of the* 12th *ACM Conference on Computer and Communications Security (CCS)* (November 2005).

Notes

[1] This limitation is especially relevant for financial and scientific applications, where a function's return value is more likely to be incorporated into the mainline calculation.

[2] Part of the CoSAK, or Code Security Analysis Kit, study found that most vulnerabilities in a set of popular open source software occur within six function calls of an input system call. If one considers a layer or two of application-internal processing and the existing (but seldom thought of from an application developer's standpoint) multiple layers within C library functions, this number makes sense.

Dynamic Spyware Analysis

Manuel Egele, Christopher Kruegel, Engin Kirda
Secure Systems Lab
Technical University Vienna
{pizzaman,chris,ek}@seclab.tuwien.ac.at

Heng Yin
Carnegie Mellon University and College of William and Mary
hyin@ece.cmu.edu

Dawn Song
Carnegie Mellon University
dawnsong@cmu.edu

Abstract

Spyware is a class of malicious code that is surreptitiously installed on victims' machines. Once active, it silently monitors the behavior of users, records their web surfing habits, and steals their passwords. Current anti-spyware tools operate in a way similar to traditional virus scanners. That is, they check unknown programs against signatures associated with known spyware instances. Unfortunately, these techniques cannot identify novel spyware, require frequent updates to signature databases, and are easy to evade by code obfuscation.

In this paper, we present a novel dynamic analysis approach that precisely tracks the flow of sensitive information as it is processed by the web browser and any loaded browser helper objects. Using the results of our analysis, we can identify unknown components as spyware and provide comprehensive reports on their behavior. The techniques presented in this paper address limitations of our previous work on spyware detection and significantly improve the quality and richness of our analysis. In particular, our approach allows a human analyst to observe the actual flows of sensitive data in the system. Based on this information, it is possible to precisely determine which sensitive data is accessed and where this data is sent to. To demonstrate the effectiveness of the detection and the comprehensiveness of the generated reports, we evaluated our system on a substantial body of spyware and benign samples.

1 Introduction

An important security threat that affects many Internet users today is spyware [24, 25]. Spyware is malicious software that attempts to silently monitor the behavior of users, record their web surfing habits, or steal their sensitive data such as passwords. Typically, the collected information is sent back to the spyware distributor, where it is (ab)used for targeted advertisement or marketing studies. This is different from other types of malware, such as viruses and worms, which generally aim to propagate to other systems and cause damage.

As the spyware problem has intensified, a number of commercial solutions have been introduced that aim to identify and remove undesired spyware. These tools are similar to anti-virus products in that they identify *known* instances of spyware by comparing the binary image of unknown samples to a database of signatures. Often, these signatures are generated manually by analyzing known spyware samples (which is a tedious task when one considers that hundreds of new samples have to be analyzed every day). Unfortunately, spyware detection tools suffer from the known drawbacks of signature-based detectors, such as the continuous need for updates of the signature database and the inability to identify previously unknown samples. Note that a major drawback of signature-based techniques is that they are also often not able to deal with simple obfuscation techniques [3].

Because signature-based detection techniques have significant shortcomings, we previously presented a behavior-based spyware detection technique that used a combination of static and dynamic analysis to identify malicious behavior of Internet Explorer browser helper objects (BHOs) [14]. Using our previous tool, we could classify unknown components as malicious or benign. Unfortunately, our approach also has a number of limitations. First, we could only assert the *possibility* that sensitive information is leaked, but we were unable to establish *exactly what* data is collected by a spyware component. This information is required by human spyware analysts that need to understand and estimate the damage caused by a specific spyware program. Second, because of our substantial reliance on static analysis of potentially hostile code, a spyware author who is aware of our technique can use code obfuscation to attempt to evade detection (or to make detection more difficult and costly).

In this paper, we present a novel dynamic analysis approach that precisely tracks the flow of sensitive information as it is processed by the web browser and any

loaded BHOs. Based on the results of our analysis, we can classify unknown components as benign programs or spyware and provide comprehensive reports on their behavior. To identify information flows, we make use of dynamic taint analysis, which tags sensitive data elements and tracks their use as they are processed. Our taint analysis combines the traditional whole system approach (in which data is tainted at a physical level) with the ability to monitor activity within individual Windows operating system processes. This is necessary to distinguish between the use of sensitive information by the Internet Explorer and the abuse of the same data by malicious browser objects. The techniques presented in this paper address limitations with our previous approach and significantly improve the quality of our analysis reports. By tracking actual information flows, we can more precisely understand and characterize the behavior of spyware. In particular, we can determine *which* sensitive data is leaked and *where* it is sent.

The main contributions of this paper are the following:

- We introduce a dynamic analysis technique to precisely monitor the flow of sensitive data as it is processed by the web browser and browser helper objects. By tracking the actual information flow using taint analysis, we can *precisely* determine which sensitive data is collected by a spyware component. Unlike previous approaches that use dynamic taint analysis, our system not only considers data dependencies, but also control dependencies.
- We present a tool that can be used to automatically analyze the behavior of unknown BHO samples and provide comprehensive reports on their activities.
- We present experimental results on a substantial body of 21 spyware samples and 14 benign BHO samples that demonstrate the effectiveness of our approach.

2 Spyware Analysis Approach

In general, spyware refers to a category of malicious software that monitors a user's operations without her consent, typically to the benefit of a third party. Spyware exists in many forms and performs actions of different levels of maliciousness. In this paper (as well as in our previous work), we explicitly focus on spyware that exploits the hooks provided by Microsoft's Internet Explorer to monitor the actions of a user. This is done by using the browser helper object (BHO) interface. In a nutshell, browser helper objects are Windows dynamic linked libraries that are automatically loaded by the Internet Explorer when it is launched. BHOs are mostly used to extend the Internet Explorer with small, custom add-ons or utilities. Examples include helpers that block pop-ups, implement support for mouse gestures, or provide embellishments (images) for web pages. Although possible, they rarely implement more complex functionality such as multi-media extensions or Java interpreters, which are realized as Internet Explorer plug-ins. Most browser helper objects do not contain any user interface elements and work in the background, responding to browser events and user input. However, they run in the same address space as the browser and have full control over the browser's functionality.

The focus on spyware that is implemented as BHOs is justified by the fact that the large majority of spyware has a component based on this technology. This is confirmed by a recent study [26], which found that out of 120 distinct spyware programs, just under 90 used BHOs as an entry point to monitor user activity. In addition, a US CERT report [10] names BHOs as one of the most frequently used techniques employed by spyware.

In previous work, we proposed the following behavioral characterization to classify a BHO as spyware:

> "A distinctive characteristic of spyware is that a spyware component (or process) collects data about user behavior and forwards this information to a third party. Thus, a BHO is classified as spyware when it **(i)** monitors user behavior **(ii)** then leaks the gathered data to the attacker."

To determine whether an unknown component exhibits malicious behavior, we used a combination of dynamic and static code analysis techniques. The dynamic analysis identified whether a BHO calls browser functions that could be used to gather sensitive user data. The static analysis then determined whether the component contained calls that could leak this information.

Experimental evaluation demonstrated that our previous system yielded good detection results with low false positives. However, there are two significant limitations with our previous approach. One is that our approach can only identify the *possibility* that information could be leaked. We were not able to record any actual information flow. Thus, it is not possible to determine precisely *which* sensitive data is accessed or leaked. Also, it is not possible to identify precisely *where* the data is sent. Obviously, such knowledge is invaluable for a human analyst who has to manually analyze a large body of new samples every day. The second limitation is our significant dependency on static analysis, which can be exploited by a spyware author who uses code obfuscation to make it difficult to disassemble the binary [20] or hide the presence of certain function calls [5, 27]. Unfortunately, when our analysis fails to identify those function calls that are associated with malicious behavior, a spyware component is incorrectly labeled as benign. At

the same time, if a more conservative approach is used, the false positive rate increases and benign samples are falsely labeled as being malicious.

2.1 Novel Analysis Approach

To address the aforementioned shortcomings, this paper introduces a novel dynamic analysis approach. The goal of our analysis is to precisely track the flow of sensitive data as it is processed by the web browser and any loaded BHOs. By monitoring the actual information flow, we can answer the question of which sensitive data is collected by a spyware component. For example, we can determine whether the spyware only records the URLs that a user navigates to, or whether parts of the visited web pages are read as well. In addition, we can determine how this information is eventually leaked. For example, data could be sent directly over the network, or first stored in a file that is later retrieved by an independent spyware process. Moreover, some spyware components are equipped with a list of URLs. Whenever the user enters a URL, it is compared to all entries in this list. When the URL matches, the BHO triggers certain actions (e.g., display an advertisement in a pop-up window). By monitoring how sensitive information is processed, such checks can be identified. In some cases, it is even possible to reconstruct the static URL list.

Dynamic Taint Analysis. Our dynamic analysis uses tainting to track the flow of sensitive information as it propagates through the system. Tainting refers to a process in which data of interest is first labeled and then tracked as it is processed by the system. With our dynamic analysis, sensitive data such as URL and web page information is tainted. Then, we track the use of this data by the Internet Explorer and its loaded BHOs. When a BHO attempts to leak any sensitive data outside of the address space of the browser (e.g., by writing data to disk or sending it over the network), this action is recorded and the component is classified as spyware. This is because according to our definition of spyware, the leaking of sensitive information is considered malicious.

Our taint analysis takes into account both data dependencies and control dependencies. A data dependency captures the fact that the result of an operation (or assignment) depends on its source operands. However, information flows can also be introduced when the execution of an operation depends on the condition of a particular variable. In this case, there is a dependency between the result of this operation and the variable that controls whether it is executed or not. Current spyware programs can be detected by only taking into account data flow dependencies. However, it is easy to develop a spyware BHO that uses control flow dependencies to propagate tainted values in a way that is not captured by data flow dependencies (an example is shown in Section 3.2). In this fashion, tainted values can be laundered and detection is evaded. To address this threat, we believe that it is necessary to stay ahead of spyware authors and already consider control dependencies.

In addition to the precise tracking of sensitive data within the Internet Explorer, we are also interested in following this data once it has left the browser's address space. The fact that tainted information was leaked is sufficient to classify a component as spyware, but it is usually helpful for an analyst to be able to further track the information flow. For example, when data is written into the memory image of a spyware helper process, the additional information that this process later sends the data to a remote server would be valuable. To track such inter-process communication and data flows, we perform whole system analysis.

Operating System Awareness. One problem for our analysis is that it needs to distinguish between actions that are performed by the Internet Explorer and those that are performed by its browser helper objects; a problem that is complicated by the fact that the browser and its components are executing in the same process. The distinction is necessary to correctly attribute sensitive information flows either to normal browser operation or to malicious activity of a spyware component. Otherwise, it would not be possible to differentiate between the Internet Explorer writing a page to its temporary cache directory or a spyware saving the same information to a hidden log. A similar problem arises when a URL is written to the browser's history file, a normal operation performed by the Internet Explorer. To summarize, the mere fact that sensitive information is written out of the address space of the Internet Explorer is *not* sufficient to characterize a BHO as spyware. The BHO is spyware only when it initiates the sensitive information flow. To distinguish between sensitive data processed by the Internet Explorer and sensitive data processed by a BHO, our analysis requires a view that is aware of operating system processes and their loaded components.

Browser Session Recording and Replaying. A fundamental challenge faced by dynamic analysis approaches is test coverage. When exposing a component to a set of test cases, one cannot be certain that these tests cover the complete functionality. As a result, it is possible that some interesting behavior is not observed. In our context, this could lead to false negatives. To increase the coverage of our analysis, we have to ensure that we expose a BHO to realistic and sufficient user interaction. To this end, we developed a test system that records the actions of a user who is surfing the web. These actions include navigation to web pages and filling in form fields. Later, during our analysis, a recorded browser session can be

replayed to the BHO. The goal is to have the Internet Explorer visit a large number of pages with different content such that a spyware component will eventually trigger and reveal its malicious behavior. This allows us to test and classify samples without manual intervention.

3 System Design & Implementation

3.1 System Overview

Our dynamic taint analysis is built on top of Qemu [1], a generic and open source system emulator. Using Qemu's emulation of an Intel x86 system, we installed Windows 2000 as guest operating system (with no service packs). The choice of Windows and the Intel x86 architecture is motivated by the fact that our analysis focuses on spyware components that are implemented as BHOs for the Internet Explorer. An overview of the system and the analysis process is shown in Figure 1.

When an unknown BHO is analyzed, it is first installed on the guest OS. Then, the Internet Explorer is launched, loading the BHO component on startup. Also, the test generator is started. The task of the test generator is to simulate a surfing user by replaying a previously recorded browsing session. When sensitive data (such as a URL that the test generator navigates to) enters the Internet Explorer process, it is marked as tainted. From this point on, the taint engine tracks how the information is processed by the browser and the BHO. To be able to distinguish between actions by the Internet Explorer and those by the BHO, the taint engine differentiates between code that is executed by the Internet Explorer and code run on behalf of the BHO. The taint engine also monitors when (and where) tainted data exits the address space of the browser. When the Internet Explorer writes out tainted data because of regular browser activity, the flow is recorded as benign. When tainted information leaks because of activity on behalf of the BHO, the information flow is recorded as malicious. In this case, the analysis engine classifies the BHO as spyware.

To keep track of the taint status of data processed by the system, we introduced a shadow memory. This shadow memory holds one byte for each byte of emulated physical memory, and also covers the eight general purpose registers of the Intel x86 CPU. The decision to use one byte for each byte of the main memory and the registers allows us to not only record whether a certain location is tainted or not, but also to assign different taint labels to each location. This assignment is helpful in tagging data elements with different taint labels depending on their origin, or to distinguish between data that is processed by the Internet Explorer and data that was touched by the BHO. To propagate taint information, we had to extend Qemu's micro instructions accordingly.

3.2 Dynamic Taint Propagation

Data Dependencies. Tainting allows to tag data elements of interest and track their propagation throughout the system. Similar to a number of previous systems that use taint propagation [2, 6, 7, 22, 23], our taint analysis is capable of tracking *data dependencies*. To this end, the taint engine marks all bytes of the output of an operation as tainted whenever any byte of any input operand is tainted. This correctly propagates taint information in those cases in which a tainted value is used as source operand in an arithmetic or logic operation, or on the right-hand side of an assignment. Note that operand values can be either taken from processor registers or fetched from memory. Unfortunately, the propagation rule outline above does not take into account the taint status of a value that is used to calculate the *address* of an operand, as only the taint status of the operand itself is relevant. This can lead to problems when tainted input is used as an index into a table (or an array). In such cases, the result of a table lookup is not labeled as tainted, and its relationship with the input value is lost. Interestingly, such lookup operations are frequently used for converting user input (for example, to convert ASCII to Unicode characters in Windows, or to map keyboard scan codes to keystrokes in Linux). Thus, we also taint the output of an operation whenever a tainted value is involved in the address computation of a source memory operand (regardless of the taint status of the memory operand that is referenced).

Control Dependencies. A system that can handle only data flow dependencies provides a spyware author with a simple opportunity to evade detection. Figure 2 provides an example that illustrates the problem. On the left side of this figure, a code fragment is shown where two conditional branches are used to "assign" the value of the tainted variable `t` to the variable `clean`, assuming that `t` only takes on the values 'a', 'b', or 'c'. Because there is no direct data dependency between `t` and `clean`, the variable `clean` will hold the same value as `t` after the execution of the code fragment, but it is not tainted.[1] Clearly, this approach can be generalized to launder arbitrary information. To mitigate this weakness, our taint analysis also considers *direct control dependencies*. To correctly handle control dependencies, the result of an operation has to be tainted whenever the execution of this operation depends on the value of a tainted variable (e.g., when an operation is guarded by an `if`-branch that tests a tainted variable, or when an operation is executed in a case branch when the corresponding switch statement used a tainted argument). Note that the result of any such

[1] Note that this example is shown in C code, although our system operates directly on x86 binaries.

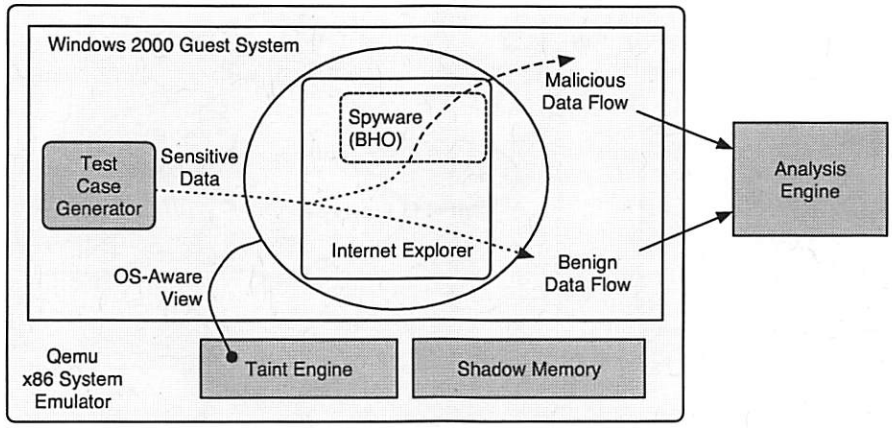


Figure 1: System Overview.

operation is tainted independently of the taint status of the source operands. Revisiting the example shown in Figure 2, and using a system that can track direct control dependencies, we observe that variable `clean` will be tainted whenever `t` is tainted. This is because the execution of any assignment operation depends on the value of `t`, and thus, there are control dependencies between `t` and the results of these assignments.

To handle control dependencies, the taint engine examines all conditional branch instructions that are encountered during execution. When such an instruction has at least one tainted operand, the taint engine has to identify all instructions whose execution is conditionally dependent on the result of the branch.[2] Using an analogy from imperative programming languages, the task is to determine the scope of a conditional branch such that this scope encloses all instructions that depend on the outcome of the branch. The results of all operations that are then executed within this scope need to be tainted.

To find all instructions that belong to the scope of a branch, static analysis is necessary. The reason is that we have to find the first instruction in the control flow graph that is executed independent of whether the conditional branch is taken or not. More formally, this instruction is the immediate post-dominator of the branch operation in the program's control flow graph. Intuitively, it is the point where the two possible execution paths after the branch operation merge. At this instruction, the scope of the branch statement ends, and it is no longer necessary to taint the results of all operations. To find this instruction, two (or more) possible execution paths need to be explored. This can only be done statically, because there is only a single path executed dynamically. As an example of a branch instruction with its corresponding scope and post-dominator node, consider the right side of Figure 2. The graph represents the control flow of the code fragment on the left. It can be seen that the last node (where 0 is assigned to `x`) is the point where the two branches of the first `if`-statement merge.

The first step in finding the instruction that ends a scope is to build a (partial) control flow graph (CFG) of the program. The control flow graph starts at the branch instruction and needs to cover all paths until the merging point. Of course, this merging point is not known *a priori*. Thus, we extend the control flow graph until we reach instructions where the disassembly process cannot continue (typically, these are function return instructions, but also indirect jumps whose target cannot be resolved statically). To build the CFG, our system uses a recursive disassembler [17]. Because we do not continue the disassembly process after instructions whose targets we cannot determine with certainty, we obtain a control flow graph that contains only instructions that are reachable during runtime. This assumes that the code is not self-modifying. Fortunately, our dynamic analysis can easily identify attempts of a BHO to modify its own code by monitoring the target of memory write operations and ensuring that no code regions are altered. Any attempt of a BHO to modify its own code is flagged as malicious.

The fact that our partial control graph is guaranteed to contain only instructions that are reachable during runtime is important, as there are a number of ways in which the attacker could attempt to thwart static analysis and the disassembly process using code obfuscation [20]. Because we use a simple analysis approach that explores paths only as long as successor instructions can be iden-

[2] Actually, the situation is a little more complicated with the x86 instruction set. The reason is that conditional jumps do not have operands themselves, but use the processor flags set by a previous compare operation to decide which branch to take. Thus, our system links the execution of an instruction that compares (or tests) tainted data with a subsequent conditional jump to identify those branches that operate on tainted data.

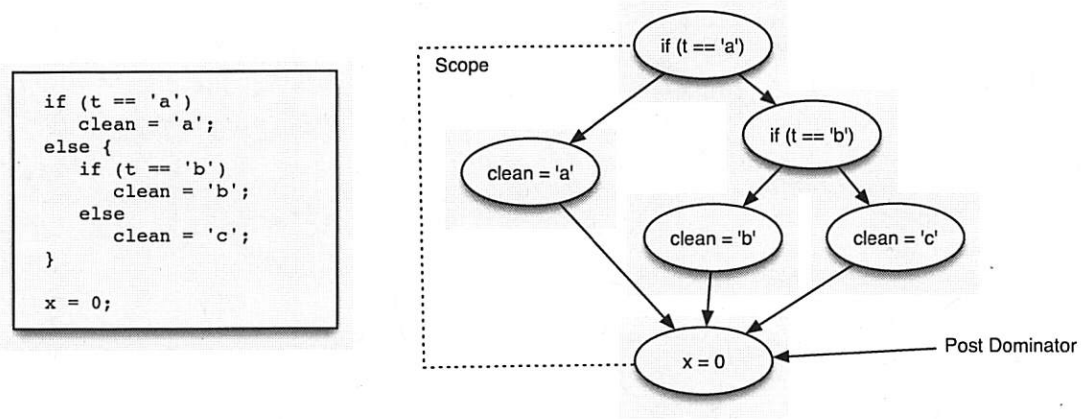


Figure 2: Control Dependency and Scope.

tified with certainty, our static analysis step is immune to these obfuscation techniques. This is a major improvement over the significantly more complex static analysis described in our previous work [14], where the complete binary is disassembled and analyzed. Of course, the control flow graph that we extract is not necessarily complete. This is typically due to the problem of indirect jump or call instructions whose targets cannot be resolved statically (e.g., in the presence of function pointers or jump tables). We recognize and handle this problem in the following step.

When the disassembler finishes, it has extracted a control flow graph that contains all instructions that are definitely reachable starting from the branch instruction. We then apply the well-known Lengauer Tarjan [19] algorithm to compute a dominator tree for this graph. This dominator tree allows us to the find the node that immediately post-dominates the branch instruction, and thus, represents the instruction that ends the scope. However, as mentioned before, the control flow graph might be incomplete. In this case, it is possible that there are multiple nodes that post-dominate the branch instruction. Hence, whenever this situation occurs, we take a safe approach and assume that the BHO contains code to thwart analysis, and label the BHO as malicious.

Note that our technique to track control dependencies is conservative, as it taints the results of all operations executed within a tainted scope. Thus, it is possible that our system introduces incorrect dependencies between variables and raises false positives. To address this issue, we only track control flow dependencies when executing code inside the BHO. The rationale is that the attacker can only control the BHO, and we assume that the Internet Explorer itself does not contain code that deliberately attempts to hide data dependencies via the control flow. Also, observe that our static analysis is only invoked when the dynamic analysis actually encounters a conditional branch instruction with tainted operands.

Previous systems that use data tainting were not able to take into account control dependencies because this conservative propagation policy typically resulted in too many tainted values (a phenomenon often referred to as taint label explosion). The fact that we only track control flow dependencies when executing code inside the BHO is very helpful to ensure that our system does not suffer from this problem. In addition, we observed in our experiments that tainted data is only used very rarely in control flow decisions, further mitigating the problem of label explosion. However, there could be cases in which BHOs process tainted data such that many more control flow decisions are based on tainted input. An example would be a Java interpreter that executes Java code loaded by the browser. In such cases, it is likely that we also suffer from memory regions that are incorrectly tainted, leading to false positives. Fortunately, such functionality is typically not realized in BHOs.

Untainting. In addition to a mechanism that flags registers and memory locations as tainted, there must also be a way to clear their taint status. In the simplest case, a register or a memory location loses its taint status when it is overwritten with an untainted value. Immediate instruction operands (constant values) are always considered untainted. In addition, one has also to take into account constant functions, which denote code sequences that always produce the same output regardless of their input. For example, the following operation is used frequently on the Intel i386 architecture to set a register to zero.[3]

```
xor %eax, %eax; // %eax = %eax^%eax
```

[3]RISC chips, in contrast, typically provide a register that is hardwired to 0.

Because this instruction always sets the register to zero, the output should not be tainted, even when the input `%eax` is tainted. Note that another variant of the same function uses a `sub` instruction instead of the `xor`. We support simple constant functions that consist of a single `xor` or `sub` instruction. However, one needs to be aware that more complex versions of constant functions may exist that are not detected. In these cases, the system could incorrectly label certain data elements as tainted, which might result in false positives. In our experiments, however, we did not observe any problems stemming from this limitation.

3.3 Bridging the Semantic Gap

A whole system emulator, such as Qemu, only provides a hardware-level view of the guest system, including physical memory, CPU registers, and I/O device status. However, for the purpose of meaningful analysis, a view at the operating system level is necessary. In other words, we have to bridge the semantic gap between these two views. In particular, we need to address two problems: (1) identifying operating system processes, so we know when the Internet Explorer is executing; (2) distinguishing what actions are performed in the context of the BHO. These problems are not entirely trivial, especially for a closed source operating system such as Microsoft Windows.

Identifying Operating System Processes. To identify operating system processes, we leverage the mechanism that Windows uses for virtual memory management (on the x86 architecture). In particular, we make use of the fact that for the current process, the `CR3` processor register stores the physical address of its page table directory. This address is unique for all running processes. To obtain the page directory address that belongs to a process, we exploit the facts that Windows stores this address as an attribute of the `EPROCESS` structure, and that a pointer to this is always mapped to the same, well-known virtual address.

Of course, our analysis has to determine the `CR3` value for the Internet Explorer before it can execute any user mode instructions. We decided to hook the Windows system call that is responsible for creating new processes (called `NtCreateProcess`). Hooking is performed by checking the processor's instruction pointer at the beginning of each operation and comparing it to the address of the `NtCreateProcess` function. This address can be obtained from the kernel symbol table that comes with each Windows distribution, usually for debugging kernel device drivers. Whenever a new process is created by invoking `NtCreateProcess`, we check the process list for the new entry and compare its name to the program(s) that we wish to monitor.[4] When the names match, the `CR3` value is extracted and the process is monitored.

Identifying Actions in the Context of the BHO. The ability to identify Windows operating system processes allows us to distinguish the operations that are performed by the Internet Explorer from those of other processes. Unfortunately, this is not sufficient. The reason is that we also need to determine which code *within* the Internet Explorer process is run because of regular browser activity, and which code is executed on behalf of a BHO. As discussed in Section 2, this is important to correctly attribute monitored behavior either to the Internet Explorer or to one of the loaded components.

Obviously, all instructions that are located directly in the code segment of the BHO are considered to run on its behalf. However, we also wish to cover the case in which the BHO code calls another function that is located elsewhere (in the Internet Explorer or any other loaded library). To correctly identify all instructions that are executed *in the context of the BHO*, the following algorithm is used:

1. Whenever execution is transferred from the Internet Explorer to the code of the BHO, record the value of the current stack pointer. This transition is recognized by observing the execution of an instruction that is located in the code segment of the BHO. Then, goto Step 2.

2. For every further instruction, check if the current value of the stack pointer is below the value stored in Step 1. If so, the instruction is executed in the context of the BHO; else it is not, and we restart with Step 1.

The rationale behind this technique is as follows: Whenever code in the BHO is called, we record the location of the current stack frame on the stack. When the BHO itself calls other functions, additional stack frames are pushed onto the stack. Because the stack grows towards smaller addresses on the x86 architecture, the stack pointer remains below the stored stack pointer. Only when all functions have returned and the BHO invokes a return operation, the stack frame of the BHO is popped from the stack and the value of the stack pointer exceeds the one stored. One problem that complicates our approach is the presence of threads. The reason is that, for each thread, the operating system allocates a different stack region in the process' virtual address space. Thus, the value of the stack pointer is only meaningful in the context of a certain thread, and switches between

[4] To be precise, we check the process list when `NtCreateProcess` returns, because at the time the function is called, the `EPROCESS` structure does not exists yet.

threads have to be identified. To do so, we examine the current identifier of the executing thread (which is located at a well-known address in the `KTHREAD` structure) whenever execution returns from the kernel.

Based on the knowledge of which code is executed in the context of the BHO, we now have the means to differentiate between data that is written by the Internet Explorer and data that is leaked on behalf of the BHO. To this end, we extend our taint propagation policy: Whenever an instruction that is executed in the context of the BHO writes tainted data, the label of this data receives a *suspicious* flag. From now on, this data is clearly identified as sensitive data that has been processed by the BHO. Whenever tainted data with a suspicious flag is later processed by other instructions, even when these instructions are not run on behalf of the browser component, they retain their flag. Also, whenever any operand of an instruction has the suspicious flag, the output is labeled suspicious as well. Data labeled with the suspicious flag must no longer leak from the Internet Explorer process. Otherwise, the BHO is considered spyware.

Evasion. A spyware author who is aware of our technique to identify actions on behalf of the BHO might attempt to evade detection. The goal for the attacker is to leak sensitive information, but let it appear as regular browser activity. One possibility is to modify the code of the Internet Explorer such that malicious actions are performed when regular browser code is executed. Another possibility is to inject new code into the address space that our analysis does not associate with the BHO. Then, some code pointer in the Internet Explorer must be redirected to point to this injected code region. Both threats can be countered by using the fact that our analysis engine has complete control of the execution of the browser and the BHO. This allows us to ensure that only those instructions are executed that are in known code regions. To prevent a malicious component from altering the contents of legitimate code regions, we can ensure that the BHO cannot remove their memory write protection (by hooking the appropriate system call). Moreover, note that evasion is not possible for the attacker by executing a statement in the BHO that pushes the stack pointer above the limit stored in Step 1. The reason is that, in this case, the instruction following the stack pointer modification is again recognized as belonging to the code segment of the BHO. Thus, the new value of the stack pointer is saved and execution continues on behalf of the BHO.

3.4 Detection & Analysis

In this section, we discuss when information is tainted and then explain when and where the use of tainted data is suspicious.

Taint Sources. A taint source can be any part in the system that precisely defines a portion of data that we wish to track. On one hand, this can be memory locations where the hardware stores information (such as buffers that hold network packets or keyboard scan codes). On the other hand, we can taint the arguments of certain functions. Currently, we use two taint sources. One taint source is used to taint all URL strings of the pages that a users visits. This can either happen by typing the URL directly into the browser's address bar or by clicking a link on a page. The other taint source taints the information that the Internet Explorer receives in response to its requests. This includes both HTTP pages and files that are downloaded. The reason for selecting these taint sources is that we consider both the URL and the content of the page as sensitive information. Whenever this information is leaked on behalf of a browser component, this component is classified as spyware. Note that it would also be interesting to taint the data that a user enters into web forms. This would allow us to identify BHOs that attempt to steal user passwords (and other private information). Including additional taint sources is quite straightforward, and as part of our future work, we are planning to taint user input as well. To taint the URLs, we hook the `Navigate` function of the Internet explorer and taint the string argument that represents the URL. Note that the hooking of an Internet Explorer function works similar to the hooking of a system call.

To taint the data that is retrieved by the Internet Explorer, we mark the return data buffer of the Windows equivalent of the Unix `receive` system call, which is called `NtDeviceIoControlFile`. Whenever this function is invoked, we first wait until it returns and then consult the return code. When data was successfully received, the appropriate buffer is tainted. We assign different taint labels to the URL and the page data to be able to distinguish between them.

Taint Sinks. When input data becomes tainted, taint information is automatically propagated by our system. The goal is to determine whether this data is eventually used in a fashion that would reveal spyware-like behavior of a browser component. According to our definition of spyware, such behavior is present in situations where tainted data is leaked by the BHO. Recall that we are not interested in writes of tainted data in general. Only a flow of information that is explicitly labeled suspicious leads to the classification of a component as spyware To detect such flows, our system monitors the interfaces that can be used to write information out of a process for the presence of suspicious information. Currently, we monitor communication over the network, writes to the file system, accesses to the registry, and communication with other processes via shared memory. While we believe that our set of sensitive sinks is comprehensive,

it is possible that we have missed a vector that a BHO could use to leak sensitive information. However, adding additional vectors to our system is straightforward, and merely a matter of monitoring the appropriate arguments of the relevant system calls.

To monitor whether information is leaked over the network, we monitor the data buffer argument to the system call `NtDeviceIoControlFile`. This system call acts as a funnel for higher-level network calls and is responsible for receiving and sending data over both UDP and TCP. To differentiate between the different roles of `NtDeviceIoControlFile`, its first parameter must be evaluated. To check for writes to files, we monitor the `NtWriteFile` system call. Also, we hook the `NtCreateFile` function to be able to later associate the file name with the file handle that is used for file access calls. Similar hooks are inserted to monitor the system calls that are responsible for writing keys and values to the registry. Note that it is typically not sufficient to check the arguments of the `NtWriteFile` system call to cover all file accesses. The reason is that files can also be memory mapped. In this case, (parts of) the contents of a file are mapped into the virtual address space of a process. Then, the file can be accessed by regular memory read and write operations. To detect tainted data that is written into memory mapped files, the system call that performs the mapping is intercepted. Whenever a monitored process maps a file into its address space, the corresponding memory regions are recorded. On any subsequent write to these monitored ranges, our analysis can derive that a file was written. For this, it is necessary to check the target addresses of every write operation. This check is performed as part of the taint propagation logic.

Note that it might be overly conservative to consider as suspicious the fact that a BHO saves data to disk. Although we have not encountered legitimate BHOs in our experiments that write URLs or web page data to a file, it is conceivable that certain legitimate applications might do so (e.g., bookmark managers). In this case, the system could be extended so that it does not immediately report a BHO as spyware that writes to disk, but instead continues to monitor what happens to the sensitive data. When, at one point, another process accesses the file, reads the sensitive information, and sends it over the network, the BHO would be classified as spyware. Otherwise, the write would be considered benign. While this extension has not been implemented, our system already supports this kind of analysis in principle (i.e., the tainting engine can track tainted data in multiple processes, and we record which bytes are tainted in a file when sensitive data is saved).

Detailed Analysis. To improve the quality of our analysis reports, we also record in more detail how code that is executed in the context of a BHO handles tainted data. One piece of information that we are interested in is whether the BHO reads tainted data at all. If a BHO never touches any sensitive information, our confidence increases that the component is not spyware. On the other hand, if tainted data is accessed, we are particularly interested in those reads where subsequent bytes of the input are accessed. This could indicate that parts of the sensitive data are copied for further processing.

Another interesting indicator to better understand the behavior of spyware is whether the monitored component performs compare operations where one of the operands is tainted. For example, when spyware compares the current URL with its own list of interesting URLs, or when the page is scanned for the presence of certain keywords, we would expect to see a number of consecutive compare instructions with tainted operands that are executed in the context of the BHO. By recording which values are compared, it is even possible for a human analyst to derive which keywords or URLs the spyware is searching for. Deriving more information about the values that the BHO is looking for can be done especially well when an x86 string compare instruction such as `cmps` is used. In this case, the operands of the instruction point to the two complete strings that are compared. Also, we check for sequences of compare operations that refer to consecutive memory locations. This allows us to identify (some) string matching routines that perform byte-by-byte comparisons.

Automated Browser Testing. When using dynamic approaches, it is very difficult to be certain that the complete range of functionality of a component is analyzed. Thus, the number of web pages visited and the interaction during the browsing phase is an important factor for the quality of the results. Clearly, requiring the human analyst to manually visit pages and fill out forms is tedious for large test sets and also prevents the system from being integrated into an automated tool-chain for spyware analysis. To address this problem, we developed a browser testing tool that allows us to automate our analysis by mimicking the surfing behavior of users. The tool can record the web interactions of a user and later "replay" them to make the Internet Explorer visit a large number of web pages without manual intervention. It also supports user input that is inserted into form fields.

The browser automation tool consists of two components: The first component is a Mozilla Firefox extension (i.e., plug-in) that records the pages a user has visited and the input she has entered into forms. The captured data is dumped into a file so that it can be later replayed. The second component is a Microsoft Windows application that first reads the information from the capture file and then replays the surfing session to the Internet Explorer. To this end, the tool first obtains a handle to the browser. Then, it repeatedly invokes the `Navigate`

method of the browser's `IWebBrowser2` interface to visit the list of stored URLs. For every web page that is visited, the tool uses the `IHTMLDocument2` Document Object Model (DOM) interface to locate all its form elements. This allows us to automatically fill out form fields that were filled out during the recorded session (using the names of the form elements). When a form is completed, it is automatically submitted.

4 Evaluation

The goal of our system evaluation is twofold. On one hand, we wish to verify the ability of our system to classify unknown browser helper objects. To this end, we analyzed a collection of spyware and benign samples and determined the fraction of samples that were correctly identified. On the other hand, we wish to demonstrate that our system provides comprehensive reports that allow a human analyst to quickly and in detail understand the behavior exhibited by a spyware component. To this end, we selected a few samples and provide a more detailed description of our findings.

4.1 Batch Analysis

To verify the ability of our system to distinguish between spyware and benign components, we compiled a test set that contained 21 spyware and 14 benign browser helper object samples. All spyware samples were provided by an anti-virus vendor. For the benign samples, we downloaded a number of different browser helper objects from various shareware sites. Of course, we made sure that these components were indeed benign by carefully checking both anti-spyware vendor and software review web sites. The benign samples were chosen from a variety of application areas. Tables 4 and 5 in the Appendix list and describe the samples that we used during our experiments. It is often difficult to determine the name of a malware sample as these names are not unique and may vary between anti-spyware and anti-virus vendors. When naming the malware samples, we used the information we were given by the anti-virus vendor.

Using our test set, we performed a batch analysis. That is, for each sample in the set, the following steps were carried out: First, the sample is loaded into the analysis environment. Then, it is installed using the Windows `regsvr32` utility. During this installation process, BHOs register themselves with the Internet Explorer so that they are automatically loaded when the browser is launched. After that, the Internet Explorer is started and the automated test generator replays a previously recorded browsing session. For this test session, we surfed to 50 web pages. The pages were chosen from different web site categories (such as adult, news, or wall paper) to provide variety. For these categories, we decided to use the ones presented in [21], a paper in which the authors analyzed different sites for the presence of spyware. The browsing session also contains typical user interactions on the sites that might be of interest for spyware, such as Google searches for free pornography, news browsing, and visits to music sites. At the end of the test, the browser is closed, and the analysis engine analyzes the log file. If the log contains any indication that sensitive information was leaked on behalf of the component under analysis, it is classified as spyware.

The sample set we used does not contain toolbar-based spyware. This is because our automated testing infrastructure currently does not support toolbars. Toolbars introduce additional GUI elements into the web browser that are not present when the initial test session is recorded. Thus, our testing tool cannot invoke any toolbar functions that require to click on GUI elements installed by this toolbar.

Table 1 shows the results for the batch analysis of our test set. The results demonstrate that all spyware components were correctly identified. In our experiments, none of the spyware samples made use of control flow evasion techniques. Thus, all malicious BHOs can be correctly detected taking into account data dependency information only. However, writing malicious code that makes use of control flow evasion is quite simple. To demonstrate that, we developed a proof-of-concept BHO that uses a sequence of `if`-statements (similar to the code shown in Figure 2) to leak sensitive information. Using only data dependencies, this BHO is classified as benign. When control dependencies are included, our system correctly identifies the malicious data transfer.

Also, most benign samples were correctly classified. However, in accordance with the results reported in our previous paper, we found two benign samples that actually *do leak* sensitive data and thus, exhibit spyware-like behavior. In one case, closer analysis revealed that no sensitive data was sent to a third party (false positive). In the other case, however, sensitive data was indeed sent to the distributor of the BHO, although very infrequently (suspicious case). In Table 2, we show a breakdown of the different mechanisms that the analyzed spyware samples used to leak sensitive information. These results underline that spyware BHOs in the wild actually make use of a variety of techniques to send collected information back to the spyware distributor.

Performance. Even though Qemu is a fast system emulator, the complete analysis of an unknown BHO with the replaying of a browsing session can take several minutes. Thus, our system is mainly intended for analysts that have to understand and classify unknown BHOs. In addition, our tool could also be used as the analysis component in an automated spyware collection system.

	Spyware	False Negative	Benign	Suspicious	False Positive	Total
Spyware	21	0	-	-	-	21
Benign	-	-	12	1	1	14

Table 1: Results for batch analysis.

Network	File System	Registry	Shared Memory	Total
11	1	3	6	21

Table 2: Different mechanisms used by spyware to leak sensitive data.

	Min.	Max.	Average
Native Windows	0.6	2.9	1.9
Qemu	1.8	6.1	3.6
Modified Qemu	17.3	79.4	35.7

Table 3: Performance overhead.

For a more detailed overview of the incurred performance penalty, refer to Table 3. This table presents the times (in seconds) that were necessary to load web pages on our test machine (Pentium IV, 2.4 GHz with 1 GB RAM); for Windows running natively, on an unmodified Qemu emulator, and on Qemu with our modifications. These numbers show the average, minimum, and maximum load times for the web pages used in our experiments. The time required to load each page increased almost linearly with the size of the page, and the resulting work necessary for rendering. It can be seen that our system incurs an average slowdown of about a factor of ten when compared to an unmodified Qemu, with an additional factor of two when compared to native execution. In a worst-case scenario, when all memory is tainted, the slowdown could significantly increase. The reason is that all conditional branches would operate on tainted data, thus triggering the static analysis step. Fortunately, as shown by our experiments, only a small fraction of memory is typically tainted, and BHOs rarely used tainted data in control flow decisions.

Although the focus of this work was not on performance, note that this overhead could probably be improved. For example, about 30% of the overhead of our system is caused by checking, for each basic block, whether the first address corresponds to a function that represents a sensitive source or a sensitive sink. Instead of checking the instruction pointer for each basic block, the interesting code parts could be memory-protected. Whenever these code regions are later accessed, a fault is raised that can be used by our system to determine that an interesting function was called. The remaining overhead of 70% is a result of the logic that propagates the taint labels. Again, this number could be significantly reduced, for example, by selectively switching between emulation and virtualization, as discussed in [11]. The memory overhead of our system is basically constant, and dominated by the size of the shadow memory, which requires one byte for each byte of emulated physical memory. Because we reserved 128 MB of memory for Qemu, the size of the shadow memory was 128 MB as well.

4.2 Detailed Analysis

The following paragraphs describe briefly the information that is contained in our analysis reports. In addition, we discuss in more detail the false positive, the suspicious sample, and three representative spyware BHOs. This discussion underlines the richness and the level of detail of the reports that are automatically generated by our dynamic analysis.

Reports After our system has analyzed a BHO, a report is generated that describes how this BHO has handled sensitive data. For every byte of sensitive input that is accessed by the BHO, we show the value and the origin (i.e., sensitive source) of this byte. Consecutive labels are combined so that accesses to strings appear as such in the output. Of course, multiple reads of the same data are suppressed and the access is shown only once. Whenever sensitive data is used in comparison operations, we show the values and labels of those bytes involved in the comparisons, as well as the values that the input is compared to. Again, compares of multiple, consecutive labels are shown in a combined form (as discussed in Section 3.4). Finally, whenever a tainted byte is leaked via a sensitive sink, the type of the sink and the leaked bytes are displayed. In all cases, information is only displayed when the tainted data has been processed by the BHO under analysis.

In general, these reports present a significant improvement compared to our previous system [14], which could

only label a BHO as spyware or benign. Previously, a tedious and time-intensive manual process was necessary to understand why an alert was raised, for example, in case of a false positive. Furthermore, the important information about the type of data that was leaked (such as the URL, or part of the page) was not available.

False Positive. The false positive listed in Table 1 is caused by the `PrivacyBird` BHO. This component implements the client side of a privacy management standard defined by the Platform for Privacy Preferences Project (P3P). The P3P standard specifies a mechanism for users to control the disclosure of their personal information on web pages. To this end, the `PrivacyBird` BHO has to retrieve a privacy policy file (which is located at w3c/p3p.xml) for every web page. To determine the server that hosts the privacy file for the current page, the BHO reads the URL and extracts the domain name. Then, the domain name is combined with the static path to the privacy file. The resulting URL is then used to fetch the privacy file. Because this contains a part that is tainted (the domain string), we detect a malicious information flow. Our analysis shows that the BHO reads the URL of every page that is visited. In addition, we can quickly confirm that for every "malicious" request, the server that is contacted is equivalent to the domain string that is tainted in this request. This information allows a human analyst to gain confidence that the `PrivacyBird` component is indeed not sending any sensitive information to a third party. Note that it would also be possible to specify a policy that classifies as benign all information flows in which information about a URL or a page is transferred back to the host from which they are loaded. The reason is that in such cases, no sensitive information is revealed to a third party. When this policy were in effect, the `PrivacyBird` BHO would not have raised a false positive.

Suspicious Sample. The suspicious sample was the `LostGoggles` BHO, a component that embellishes Google search results by adding pictures to the search hits. To this end, the BHO downloads a piece of JavaScript code from the author's web server when a search request is sent to Google for the first time. Subsequently, this JavaScript snippet is inserted into every result page returned by Google. When the script is downloaded, the BHO sets the referrer header in the HTTP request that fetches the JavaScript file. This referrer header contains the URL of the Google search that was issued before. Thus, `LostGoggles` does leak possibly sensitive user information, although the data is probably sent inadvertently and only once when the script is obtained.

Interestingly, both the web pages of `PrivacyBird` and `LostGoggles` emphasize that the components are not spyware, even though they do send information over the network (which is behavior typically associated with spyware). Using our tool, a detailed analysis can help to provide more evidence to decide whether a component is using data as described.

Spyware Samples. `Zango` advertises its products (such as games or screen savers) as ad-supported freeware. During our analysis, we determined that the `zangohook.dll` BHO, which is shipped with the company's instant messaging client, is spyware. More precisely, our system detects that whenever a web page is visited, the BHO reads the current URL and copies it to a previously opened shared memory section. From this shared memory section, the data is later read by the spyware helper process `zango.exe`. The `Zango` example underlines the importance of monitoring shared memory areas that can be used by a BHO to write out data to other processes. Also, it demonstrates the usefulness of whole system analysis, which allows us to follow the sensitive data to the spyware helper process.

The `e2give` BHO reads the URL of every site that is visited and compares it to a list of URLs stored in the BHO. This check is implemented by consecutively matching the current URL against every item in the BHO's URL list. As our analysis checks for compare instructions that involve tainted operands, the log file contains the complete list of URLs that the BHO checks against. If any of the requested sites is found in the list, the BHO redirects this request to a different server. In this case, the original URL is passed as an argument to the redirected GET request. This constitutes a flow of sensitive data that is correctly identified by our system. The `e2give` BHO is interesting for two reasons. First, it demonstrates the ability of our tool to extract lists of URLs that a spyware monitors. Second, it underlines the importance of test coverage. When none of the URLs in the BHO list were visited, the sample would be misclassified as benign (as sensitive data is only leaked in case of a match).

Finally, our analysis detected that the `stdup.dll` BHO (i.e., Borlan) submits the URLs of all visited pages to a remote server. Thus, the sample was classified as spyware. This BHO is interesting because a scan with the latest versions (at the time of writing) of the commercial anti-spyware tools AdAware [18] and SpyBot [15] yields no detection. This demonstrates that our analysis is capable of detecting previously unknown spyware samples.

5 Related Work

Malware Detection. To combat the increasing spread of spyware, a number of commercial solutions have been developed. For example, both AdAware [18] and SpyBot [15] are popular tools that are able to remove a large

number of spyware programs. The problem with existing spyware detection tools is that they use signatures to detect known spyware instances. Thus, they require frequent updates to their signature database and cannot identify previously unseen samples.

To address the limitations of signature-based malware detection, researchers have recently proposed behavior-based techniques. These techniques attempt to characterize a program's behavior in a way that is independent of its binary representation. By doing this, it is possible to detect entire classes of malware. An example of using behavior characterization to detect malicious code is Microsoft's Strider Gatekeeper [26]. This tool monitors auto-start extensibility points (ASEPs) to determine if software that will be executed automatically at startup is being surreptitiously installed on a system. In [4], the authors characterize different variations of worms by identifying semantically equivalent operations in the malware variants. A similar approach is followed in [16], where the behavior of kernel-level rootkits is modeled.

In a previous paper [14], we introduced a behavioral approach to detect spyware. For that paper, we used the same underlying characterization of spyware as in this work (that is, a BHO is considered spyware when it leaks sensitive information). However, the analysis techniques are completely different. For the former paper, we mainly relied on binary, static analysis to identify code paths in the BHO that can leak information. A small dynamic component was used to find the entry points for the static analyzer. In this paper, we developed (from scratch) a dynamic taint analysis system that supports data and control dependencies and is operating-system aware. Using our new system, we can automatically generate rich reports that precisely identify which sensitive information a BHO touches and where it is eventually stored. This was not possible with our previous system. Also, we removed our reliance on complex binary static analysis, which is vulnerable to code obfuscation and evasion.

Virtual Machines and Taint Analysis. For this paper, we use a virtual machine (Qemu) to monitor the behavior of unknown browser helper objects. This has the benefit that our analysis runs in complete isolation from the samples that are examined, making it much harder for spyware to detect the presence of our system. Other researchers made similar use of virtual machines to detect and prevent intrusions [9, 12] and to analyze attacks [8, 13]. Also, virtual machines have been used to implement whole system analysis based on dynamic tainting. For example, a system was proposed in [2] to use taint information to track the lifetime of data. The goal was to determine the use of sensitive information by the operating system and large applications. Other researchers used taint analysis to monitor program execution for the use of tainted data as arguments to control flow instructions or systems calls [6, 7, 22, 23] (a system to perform taint propagation particularly efficient was presented in [11]). The aim of these systems is to identify exploits at runtime, and, in some cases, to create signatures for detected attacks. There are a number of differences to our work. First, we analyze malicious code that can be deliberately designed to thwart detection. Thus, it was necessary to extend our taint analysis with the capability to handle control dependencies in addition to data dependencies. Second, previous systems focus on whole system emulation only and can neither distinguish between operations performed by different operating system processes (and individual components of these processes) nor keep track of which component has accessed sensitive data. Finally, the aim of previous systems is to detect exploits, while the goal of our system is to identify spyware components and comprehensively analyze and document their behavior.

6 Conclusions

In this paper, we presented a novel dynamic analysis approach to classify unknown browser helper objects and capture their behavior. The goal of our system is to automatically identify spyware that is installed in the form of browser helper objects for the Microsoft Internet Explorer. To this end, we monitor the way that the Internet Explorer and installed browser helper components handle sensitive user information (such as the URL that a user visits or the content of the web pages that are loaded). A BHO is classified as spyware when it leaks sensitive information outside of the browser process. In addition to classification, the analysis also provides a rich and comprehensive description of the actions performed by BHOs. The experimental results on a substantial body of spyware and benign samples demonstrate the effectiveness of our approach.

Acknowledgments

We would like to thank our shepherd Andrew Warfield and the anonymous referees for their valuable feedback. This work was supported by the Austrian Science Foundation (FWF) under grant P18157, the FIT-IT project Pathfinder, and the Secure Business Austria competence center.

References

[1] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *Usenix Annual Technical Conference, Freenix Track* (2005).

[2] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding Data Lifetime via Whole System Simulation. In *Usenix Security Symposium* (2004).

[3] CHRISTODORESCU, M., AND JHA, S. Testing Malware Detectors. In *ACM International Symposium on Software Testing and Analysis (ISSTA)* (2004).

[4] CHRISTODORESCU, M., JHA, S., SESHIA, S., SONG, D., AND BRYANT, R. Semantics-Aware Malware Detection. In *IEEE Symposium on Security and Privacy (Oakland)* (2005).

[5] COLLBERG, C., THOMBORSON, C., AND LOW, D. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Conference on Principles of Programming Languages (POPL)* (1998).

[6] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-End Containment of Internet Worms. In *20th ACM Symposium on Operating Systems Principles (SOSP)* (2005).

[7] CRANDALL, J., AND CHONG, F. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *37th International Symposium on Microarchitecture (MICRO)* (2004).

[8] DUNLAP, G., KING, S., CINAR, S., BASRAI, M., AND CHEN, P. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2002).

[9] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Network and Distributed Systems Security Symposium* (2003).

[10] HACKWORTH, A. Spyware. US CERT Publications, 2005.

[11] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical Taint-based Protection using Demand Emulation. In *EuroSys Conference* (2006).

[12] JOSHI, A., KING, S., DUNLAP, G., AND CHEN, P. Detecting past and present intrusions through vulnerability-specific predicates. In *Symposium on Operating Systems Principles* (2005).

[13] KING, S., AND CHEN, P. Backtracking Intrusions. In *Symposium on Operating Systems Principles (SOSP)* (2003).

[14] KIRDA, E., KRUEGEL, C., BANKS, G., VIGNA, G., AND KEMMERER, R. Behavior-Based Spyware Detection. In *Usenix Security Symposium* (2006).

[15] KOLLA, P. Spybot Search & Destroy. `http://www.safer-networking.org/`, 2006.

[16] KRUEGEL, C., ROBERTSON, W., AND VIGNA, G. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Annual Computer Security Applications Conference (ACSAC)* (2004).

[17] KRUEGEL, C., VALEUR, F., ROBERTSON, W., AND VIGNA, G. Static Analysis of Obfuscated Binaries. In *Usenix Security Symposium* (2004).

[18] LAVASOFT. Ad-Aware. `http://www.lavasoftusa.com/software/adaware/`, 2006.

[19] LENGAUER, T., AND TARJAN, R. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS) 1*, 1 (1979).

[20] LINN, C., AND DEBRAY, S. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM Conference on Computer and Communications Security (CCS)* (2003).

[21] MOSHCHUK, A., BRAGIN, T., GRIBBLE, S., AND LEVY, H. A Crawler-based Study of Spyware on the Web. In *Network and Distributed Systems Security Symposium (NDSS)* (2006).

[22] NEWSOME, J., AND SONG, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium (NDSS)* (2005).

[23] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *ACM SIGOPS EUROSYS* (2006).

[24] SAROIU, S., GRIBBLE, S., AND LEVY, H. Measurement and Analysis of Spyware in a University Environment. In *Usenix NSDI* (2004).

[25] THOMPSON, R. Why Spyware Poses Multiple Threats to Security. *Communications of the ACM 48*, 8 (2005).

[26] WANG, Y., ROUSSEV, R., VERBOWSKI, C., JOHNSON, A., WU, M., HUANG, Y., AND KUO, S. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management. In *Usenix Large Installation System Administration Conference (LISA)* (2004).

[27] WROBLEWSKI, G. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, 2002.

Appendix

Sample Name	Description
McAfeeAntiPhishingFilter	Antiphishing solution
AT&T P3PClient	Privacy utility
LostGoggles	Search enhancing utility
Earthlink Toolbar	Scam blocker
PopUpBlocker	Utility to block popups
CookiePal	Cookie management
SpywareGuard	Spyware protection utility
ezSaveFlash	Utility to save flash files
keepit	File management utility
KillaFing3	Utility to block popups
Super Popup Blocker	Utility to block popups
SAPplayer	Plays music/video files
BookmarkBuddy	Bookmark management
Plug-In for blind users	Render page for blind users

Table 4: Benign samples.

Sample Name	Description
ZangoIM	Universal instant messaging
BargainBuddy	Bundled Spyware
RAX Search Helper	Search tool
Sitestep	Travel price comparison
Borlan (stdup.dll)	Targeted ads
HtmlEdit Module	Targeted ads
Clear Search	Search tool
IEHelper Module	Installs third-party components
Generic BHO module	Targeted ads
eUniverse	Targetede ads
eZula	URL collector
W01	URL collector
Adware.MediaPlaceTV	Targeted ads and url collector
msnetwrk	URL collector
hopster	URL collector
Replace module	URL collector
e2Give	URL collector
Generic data miner	URL collector
Adware-Click	Targeted ads
CWSMeup-B	Search string collector
HuntBar BHO	URL collector

Table 5: Spyware samples.

Evaluating Block-level Optimization through the IO Path

Alma Riska
Seagate Research
1251 Waterfront Place
Pittsburgh, PA 15222
Alma.Riska@seagate.com

James Larkby-Lahet
Computer Science Dept.
University of Pittsburgh
Pittsburgh, PA 15260
jamesll@cs.pitt.edu

Erik Riedel
Seagate Research
1251 Waterfront Place
Pittsburgh, PA 15222
Erik.Riedel@seagate.com

Abstract

This paper focuses on evaluation of the effectiveness of optimization at various layers of the IO path, such as the file system, the device driver scheduler, and the disk drive itself. IO performance is enhanced via effective block allocation at the file system, request merging and reordering at the device driver, and additional complex request reordering at the disk drive. Our measurements show that effective combination of these optimization forms yields superior performance under specific workloads. In particular, the impact on IO performance of technological advances in modern disk drives (i.e., reduction on head positioning times and deployment of complex request scheduling) is shown. For example, if the outstanding requests in the IO subsystem can all be accommodated by the disk queue buffer then disk level request scheduling is as effective as to close any gaps in the performance between IO request schedulers at the device driver level. Even more, for disk drives with write through caches, large queue depths improve overall IO throughput and when combined with the best performing disk scheduling algorithm at the device driver level, perform comparably with an IO subsystem where disks have write-back caches.

1 Introduction

The IO hierarchy has grown long and complex as its main goal is to close as much as possible the performance gap between memory and disk drives. While this gap has remained significant, the amount of resources added in the IO path has increased and allows for advanced optimization in various levels of the IO path. In this paper, we take a look at the effectiveness of various optimization techniques applied at main components of the IO path such as the file system, the device driver scheduler, and the disk drives themselves. In particular, our focus is reordering of IO activity throughout the IO subsystem to improve IO performance.

Reordering of the IO work is non-work conserving because the overhead of disk head positioning associated with each disk request is different with different request schedules. Hence it becomes critically important to order the same set of requests such that the overall IO work is minimized. Early on, disk scheduling aimed at minimizing the linear disk head movement (i.e., seeks) [3, 4, 7, 20] and later evolved to minimizing the overall head positioning phase of a disk request service [1, 10, 21]. Request reordering can take place effectively only at certain layers of the IO path, which is commonly composed of a device driver, an array controller, and multiple disks that communicate with each-other through interfaces such as SCSI. For example, seek-based request reordering is traditionally done at the device driver and/or array controller, while position-based request reordering can only be done at the disk level where the accurate information about head's position is available.

In addition to effective reordering of IO requests, IO optimization techniques aim at reducing the number of requests sent down the IO path, by exploiting the temporal and spatial locality in the stream of requests and merging consecutive ones. Request merging enhances IO performance because it reduces the overall head positioning overhead which is associated with each request and it is independent of the request size. Actually, the current default disk scheduler for Linux is Anticipatory [9], which even waits idle, if necessary, to fully explore the sequentiality in the stream of IO requests.

In this paper, we focus on identification of the IO path layers, where specific IO optimization techniques are effective. For this, we conduct measurements in a system whose IO hierarchy consists of the file system, device driver, and the disk itself. Experimenting with an array controller in this path did not indicate qualitatively different results and we opted not to include it in the IO path for the bulk of the experiments. The effectiveness

of IO optimization is evaluated at the application level by quantifying the effect that request merging and reordering at different IO layers have on overall system performance.

We find that effective request merging is the key approach to achieve the maximum throughput in the IO subsystem, under heavy load. Although such optimization happens at the device driver level, via scheduling, its effectiveness is determined at the file system level. Specifically, the Reiser file system sends down more requests than all other file systems evaluated, but after merging at the device driver results to the smallest number of disk requests. Consequently, ReiserFS achieves the highest application throughput, which for a specific workload, is double the throughput of other file systems.

Our measurements show that disk level optimization in the form of scheduling is particularly effective and, although, disks are at the end of the IO path, it effects the overall application throughput. Specifically, if the load in the IO subsystem is medium, (i.e., a scenario that diminishes the importance of request merging) then by scheduling effectively at the disk level, one can close the application-level performance gap resulting from ineffective scheduling at the device driver. The effectiveness of disk level scheduling is also noticeable when comparing write-back and write-through disk cache policies. While write-back is better performing with low disk queue depths, write-through gains the performance difference as the disk queue depth is increased, which makes the latter a more attractive alternative given the enhanced data reliability and consistency it provides.

The rest of this paper is organized as follows. In Section 2, we describe the measurement environment. Section 3 explains and evaluates the applications, that we run to generate IO load. We analyze the file system level of the IO path in Section 4 and the device driver level in Section 5. The disk drive level of the IO path is evaluated in Section 6. We discuss related work in Section 7. Section 8 concludes the paper by summarizing our findings.

2 Measurement Environment

Our intention is to analyze the effectiveness of IO work optimization throughout the IO path, which (in our case) consists of the file system, the device driver disk scheduler, and a single disk drive. While we did not conduct measurements with multi-disk systems, we measured a system where the disk was attached to an array controller. The presence of an array controller (i.e., more resources and another level of scheduling) did not affect our results qualitatively and is not discussed here in detail for sake of presentation clarity.

We conduct measurements in a system (i.e., Dell Power Edge 1750) that runs the Postmark benchmark [11] on top of a Linux operating systems (i.e., Gentoo Linux 2.6.16-git11 distribution). Postmark is configured for four different scenarios as described in Section 3, that generate IO workloads with different characteristics. Postmark loads the system heavily. Building the Linux kernel multiple times simultaneously serves as our second application which loads the system with a range of load levels.

We also evaluate four file systems, (Ext2, Ext3, ReiserFS, and XFS), four disk scheduling, (No-Op, Deadline, CFQ, and Anticipatory), and three different SCSI Seagate disk drives (Cheetah ST318453LC, ST3146854LC, and ST3300007LC which we refer to as the 18 GB, 146 GB, and the 300 GB disks, respectively). Table 1 summarizes the details of our measurement testbed, while further details for each component, such as the file systems, device driver schedulers, and disk drives parameters, are given in the corresponding sections later on. Unless otherwise stated, the disks in our measurement testbed have write-through cache enabled.

System	Dual Intel Xeon CPU 2.40GHz, 1GB memory, LSI Fusion MPT SCSI Controller
OS	Gentoo Linux 2.6.16-git11
Application	Postmark / Linux kernel build
File System	Ext2, Ext3, ReiserFS, XFS
IO scheduler	No-Op, Deadline, CFQ, Anticipatory
Disk Drive	18 GB/15Krpm, 146 GB/15Krpm, 300 GB/10Krpm

Table 1: Specifications of the measurement system.

The `blktrace` tool that comes with the Linux 2.6 kernel module is used to trace the IO activity at the device driver level. The data obtained is further parsed using `blkparse`. Tracing using the `blk` tools captures the entire activity at the IO device driver and includes *queuing* a request, *merging* it with an already queued request (if predefined sequentiality criteria holds), *dispatching* it to the SCSI interface, and handling its *completion*. In addition to collecting and processing data via the `blk` tools, we conduct our own data post-processing to identify the work incoming to the device driver from the file system and the work outgoing from the device driver and completed by the disk. In our measurement system, the working set is located in a different disk from the disk that hosts the OS. Also the data collected via `blktrace` was sent through the network and not stored in the local machine to minimize effects to the targeted SCSI bus.

All our measurements are conducted on clean file systems. We argue that our conclusions hold in the case of an aged file system, because mostly our evaluation is related to the workload characteristics within the working set. Once the working set is stationary over a period of time then optimization depends mostly on the

working set size, request interarrival times, and randomness/sequentiality of workloads. With Postmark and Linux kernel builds, we cover a range of workloads with respect to working set size, request interarrival times, and randomness/sequentiality and expect the aged file system behavior to fall in any of the above evaluated categories.

We measure the IO subsystem performance via the application throughput. We chose this measure of interest, because our goal is to evaluate the overall effectiveness of combining various optimization techniques at different levels of the IO path.

3 Application Layer

Our first application is Postmark [11], which benchmarks the performance of e-mail, netnews, and e-commerce classes of applications. Postmark works on a pool of changing files, (i.e., the working set), and generates an IO-bound write-dominated workload. Because Postmark heavily loads the storage subsystem, it is our benchmark of choice for evaluating optimization efficiency in the IO path.

Postmark generates an initial working set of random text files ranging in size from a configurable low bound to a configurable high bound. The range of file sizes determines Postmark's workload sequentiality because Postmark's activity in each file is proportional to its size. Postmark's working set is also of configurable size by specifying the number of files in it. Postmark starts with creating the working set (first phase). A number of *transactions* are executed (second phase). Finally, all files are deleted (third phase). Each Postmark transaction consists of a pair of smaller transactions, which are *create file* or *delete file* and *read file* or *append file*. Postmark's throughput, used here as a measure of system performance, is the number of transactions per second during the *second* phase of the benchmark execution, which represents another reason why an aged file system with similar working set and workload characteristics should behave similarly.

Work load	File Size	Work Set	File size	No. of Files	Transactions
SS	Small	Small	9-15 KB	10,000	100,000
SL	Small	Large	9-15 KB	200,000	100,000
LS	Large	Small	0.1-3MB	1,000	20,000
LL	Large	Large	0.1-3MB	4,250	20,000

Table 2: Postmark specifications for each workload for the 18 GB disk. Specifications hold for the other two disks, except that for the LL workload 40,000 transactions are completed.

We benchmark four different Postmark workloads by varying the low and high bounds of file sizes and the number of files in the working set. We aim at having *small* working sets, i.e., occupying a few percentage of the available disk space and *large* working sets, i.e., occupying 25% for the largest capacity disk (i.e, the one with 300 GB) to approximately 50% for the smaller capacity disks ((i.e, the 18 GB and the 146 GB ones). The working set size affects the amount of seeking that is associated with each transaction. The working set size is controlled via the number of files.

Although Postmark generates a workload that randomly accesses the files within the working set (see Figure 1), the disk-level randomness depends on the average file size in the working set. We get a more random workload by setting the file size boundaries to be only a few KBytes and a more sequential workload by setting the file size boundaries to a few MBytes. Table 2 describes in detail how Postmark is configured to generate four different workloads. Throughout this paper, Postmark is configured with its buffer parameter set. Exploring the cache impact on IO performance, although important, is not addressed here.

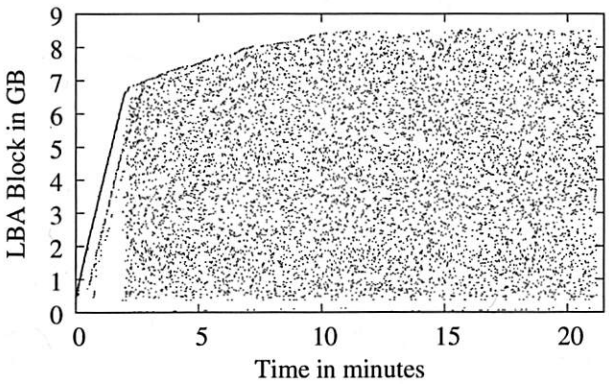
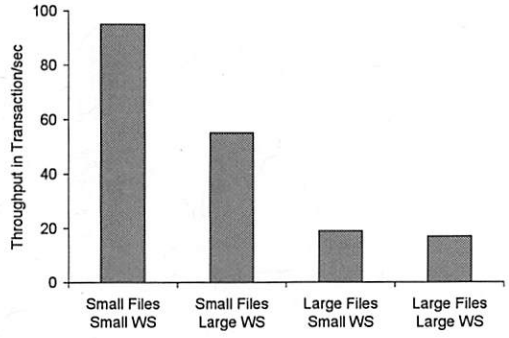


Figure 1: Access patterns for the LL workload in Table 2. The system configuration is: Ext3, anticipatory, and the 18 GB disk.

Figure 1 shows the disk access pattern for the LL workload of Table 2, by plotting the start LBA of each request on the y-axis as a function of request arrival time. The other workloads have similar access patterns and are not shown here for sake of brevity.

During the file creation phase, the disk access pattern is mostly sequential, which in the plot appears as straight lines. The sequentiality in this phase is one of the reasons that later on the read and write average request size is (in some cases) larger than the maximum file size in the working set. For the file access phase of the benchmark, the access pattern is random within the working set, which appears in the plots as a block of numerous dots (each marking a single disk request). The SS and LS workloads are localized and entail less disk head po-



Workload	SS	SL	LS	LL
IOPS	424.62	310.87	278.68	227.91
Bandwidth	60.61	62.15	228.19	269.80

Figure 2: The plot depicts Postmark's throughput in transactions per second for the four workloads of Table 2. The system configuration is: Ext3, anticipatory, and the 18 GB disk. The table gives disk IOPS and bandwidth.

sitioning overhead than for the SL and LL workloads. The large working sets that span over 50% of the available space (for the 18 GB and 146 GB disks) experience more seeking overhead. Also the sequentiality in workloads increases with the file size in the working set.

Figure 2 plots the Postmark throughput, measured in transactions per second, during the file access phase of the benchmark, for each of the four workloads of Table 2. The measurement testbed is configured with its default setting, i.e., Ext3 file system, Anticipatory disk scheduling algorithm, and the 18 GB disk drive with a queue depth of 4. The highest Postmark throughput is achieved for the SS workload. This is expected because Postmark's transaction size is linear to the file size. Consequently, workloads with large files have longer transactions than the workloads with small sizes. Transaction length affects Postmark throughput by as much as 5 times (i.e., the difference between throughput under the SS workload and throughput under the LL workload.

A similar trend is captured also by the IOs per second (IOPS) shown in the table of Figure 2. Again, the SS workload achieves the highest number of disk requests completed in each second. Between the workloads with the same file sizes, the difference in Postmark throughput (and IOPS) is related to the seeking through different file locations. For small working sets the seeking is shorter than for large working sets. The seeking overhead reduces Postmark throughput by as much as 50%. LS and LL workloads perform similarly with regard to Postmark throughput because their sequentiality (lack of randomness) causes the optimization down in the IO path to be similarly (in)effective for both of them. This is not the case for the random SS and SL workloads.

A second application that we use to benchmark the effectiveness of the optimization in the IO path is building multiple Linux kernels simultaneously. The Linux kernel builds yield similar access patterns to the disk as Postmark (when it comes to randomness) but the request interarrival times are longer and the working set spans in bands over some area of the disk. We use this second benchmark to evaluate how effective request merging and reordering becomes at the device driver and the disk drive when the system is loaded moderately.

4 File System Level

In our evaluation, we conduct measurements in four different file systems, namely, Ext2, Ext3, ReiserFS, and XFS, which are supported by any Linux distribution. The main characteristics of these file systems are summarized in the following:

- **Ext2** is a standard FFS-like file system, which uses *cylinder groups* for data placement and single, double, or triple indirect metadata blocks.
- **Ext3** is also an FFS-like file system, whose data structures are backward compatible with Ext2. However, Ext3 uses a special file as a journal to enhance file system consistency and data reliability.
- **Reiser** file system has also single contiguous journal and uses a B^+-tree as its metadata structure.
- **XFS** uses also a single contiguous journal, as well as, allocation groups, and extent-based B^+-tree for its metadata management.

The throughput of the four different workloads described in Table 2 under the four file systems in our study is shown in Figure 3. The device driver scheduler is Anticipatory, and the queue depth at the 18 GB disk is 4. Under the more sequential workloads, as it is the case of the LS and LL workloads with large files, the difference in Postmark throughput between the four file systems is minimal. They behave very similarly because the sequentiality in the workload provides an almost optimal ordering with little room for further optimization at the file system level. This is not the case for the small files workloads, which have more randomness and consequently more room for optimization. In particular, we observe superior Postmark performance under ReiserFS for the SS workload with as much as twice the throughput of the other file systems.

The work in MBytes of data read and written by Postmark and processed by the file system is presented in

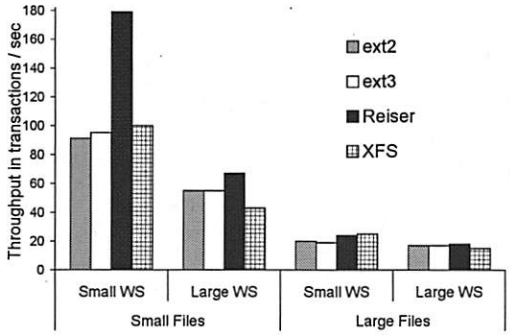


Figure 3: Postmark throughput for the four file systems and the workloads of Table 2. Measurements are done on the 18 GB disk and Anticipatory scheduling.

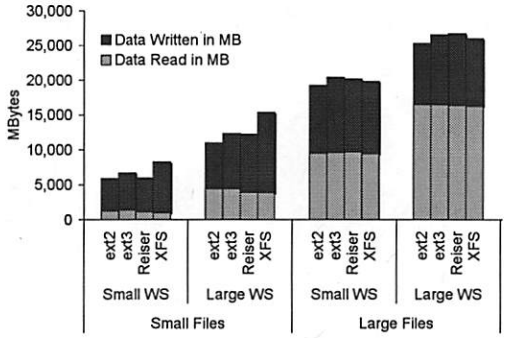


Figure 4: The amount of data read and written by each file system under the four different workloads of Table 2. Measurements are done on the 18 GB disk and Anticipatory scheduling.

Figure 4. In our testbed, we do not have a measurement point in front of the file system to exactly measure the amount of work added to the IO subsystem by the file system itself. The work processed by the Ext2 file system (i.e., no journaling) is the best approximation to the Postmark generated IO workload.

With a few exceptions, there is no substantial difference in the amount of work for either reads or writes. In particular, when comparing the amount of work under the Ext2 file system and the other three file systems that maintain a journal for metadata management. The XFS file system seems to be an outlier when it comes to the extra amount of work to manage the journal under the small files workloads (see the last bar of the first two sets of bars in Figure 4). Because the application (i.e., the scenario under the Ext2 file system) and the file system, approximately request the same amount of work (either reads or writes) to be processed by the storage subsystem, any differences in the overall application throughput (within the same workload) as reported in Figure 3 is attributed to the effectiveness of work optimization in the IO path.

By changing *only* the file system and fixing the IO workload, disk scheduling, disk queue depth, and other system parameters, the measured difference in application level performance (i.e., Postmark throughput) is attributed to the average disk request size, since we concluded that all file systems process the same amount of work in Figure 4. That is because, for the same amount of data to be transferred, the traffic composed of large disk requests has fewer requests than the traffic composed of small disk requests. Consequently the overhead associated with the disk head positioning is less for the workload composed of large requests than for the workload composed of small requests. Although, small requests get served faster than large requests, in all our measurements, a stream of few large requests always outperformed a stream of many small requests.

Figure 5 shows the average request size for the four workloads in our evaluation and the four file systems. Request size is measured in the incoming and the outgoing traffic of the device driver which corresponds, respectively, to the outgoing traffic of the file system and the incoming traffic at the disk drive. The scheduler at the device driver is Anticipatory and the disk is the 18 GB one with queue depth of 4. Generally, write disk traffic has longer requests than read disk traffic, in particularly for the workloads with large files. There is noticeable difference between the average disk request size under ReiserFS and the other three file systems. The incoming disk traffic under ReiserFS has the largest read and write request size.

While disk read and write requests under ReiserFS are of comparable size, the read and write requests outgoing from ReiserFS have different sizes. ReiserFS and the other three file systems send down the IO path requests for large reads, while only Ext2 and XFS do the same also for writes. Ext3 and ReiserFS send down the IO path constant size write requests of 4 KB relying on the device driver scheduler to merge the sequential ones. Although, Ext3 and ReiserFS do send down to the device driver, 4 KB write requests, the outgoing write requests are larger for ReiserFS than for Ext3. Overall, ReiserFS is more effective at data allocation.

The same points made in Figure 5, are reinforced with the results presented in Figure 6, where we plot the number of requests in the traffic stream incoming at the disk drive (results are similar for the stream outgoing FS). Note that while ReiserFS results consistently in the smallest number of requests sent to disk, XFS (with the exception of one case) is the second best. This indicates

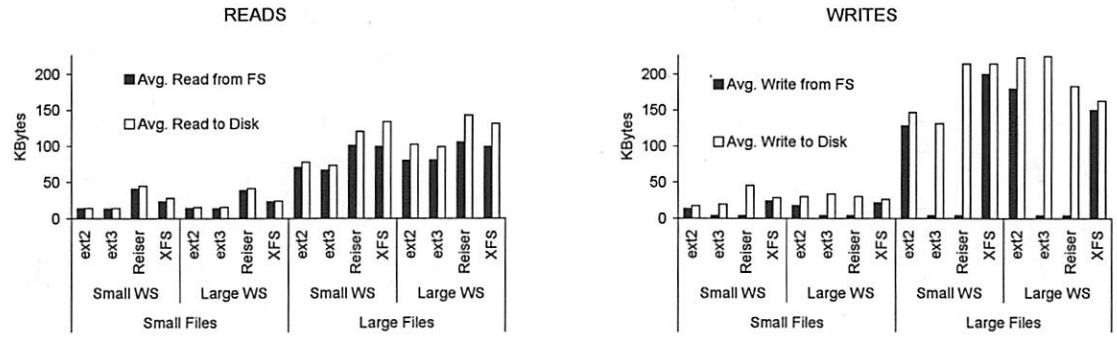


Figure 5: The average request size of reads (left) and writes (right) in the outgoing stream of requests from the file system and the incoming request stream at the disk. The four workloads of Table 2 are evaluated under four file systems. Measurements are done on the 18 GB disk and Anticipatory scheduling.

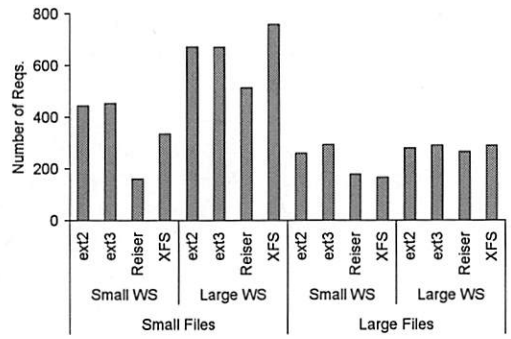


Figure 6: Number of requests in the incoming request stream at the disk, (i.e., the outgoing request stream at the device driver). The four workloads of Table 2 are evaluated for four file systems. Measurements are done on the 18 GB disk and Anticipatory scheduling.

that the constant size write requests sent to the device driver from the file system are not the reason for ReiserFS superiority. Instead, block allocation and maintaining of the journal should be considered as main factors for ReiserFS best performance for the workloads under evaluation. Although the final request merging does happen at the device driver level and, as we discuss it further in the following section, it is a characteristic that sets apart the disk schedulers at the device driver, we stress that it is the data management within the file system level that facilitates the effective outcome of request merging.

Figure 5 indicates that the main differences between the file systems are on handling write traffic rather than read traffic. The latter is mostly handled at or above the file system level (exploiting all the semantics available from the application) and also the system cache availability. Write traffic on the other hand is left to be optimized by the IO subsystem (starting with the file system). Consequently, the impact of write optimization on overall application throughput is high, because Postmark workload is write-dominated.

For the reminder of this paper, we focus mostly on the effectiveness of device driver disk scheduling and disk level queuing. To facilitate a concise presentation, we limit ourselves on measurements in one file system only and we choose ReiserFS because we identified it as the most effective file system when it comes to IO work optimization at the file system level for the workloads under our evaluation.

5 Device Driver Level

In the previous sections, we analyzed the work generated at the application level and its first optimization at the file system level. In this section, we focus on the device driver level and discuss the tools available at this level to further optimize the stream of IO requests.

At the device driver level, request scheduling is the main tool to optimize work. Ordering of requests in the elevator fashion reduces the seek overhead experienced by each request sent down to the disk. Here, we test four different disk scheduling algorithms, which are available in any Linux distribution. Three of the disk scheduling algorithms that we evaluate (namely, Deadline, Anticipatory, and Command Fair Queuing) are elevator-based and the fourth one (i.e., No-Op) is a variation of the First-Come-First-Serve algorithm (as explained below).

Apart from ordering requests, disk scheduling algorithms try to merge consecutive requests so that as few as possible are sent down to the disk. In Section 4, we discussed the importance and the impact of IO request merging on overall system performance. All evaluated disk scheduling algorithms differ in their way they merge

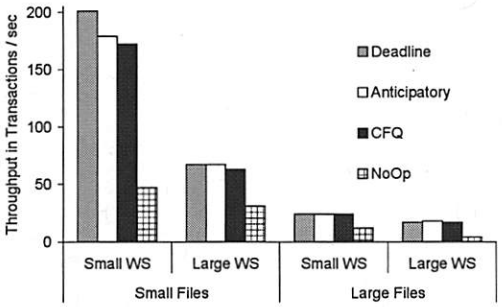


Figure 7: Postmark throughput under four different device driver schedulers and the four workloads of Table 2. Measurements are done on the 18 GB disk and ReiserFS.

requests, and their effectiveness depends on that. In the following, we describe in detail the four evaluated disk scheduling algorithms:

- **No-Op** is a first come first serve algorithm that merges sequential requests only if they arrive one after the other preserving the FCFS order.
- **Deadline** behaves as a standard elevator algorithm, i.e., it orders the outstanding requests in the order of the increased estimated seek distances, unless a read has been waiting for 300 ms or a write has been waiting for 5 seconds.
- **Anticipatory** (default) is the same as Deadline, when it come to ordering requests based on their seek distance. However sometimes it pauses for up to 6ms, in order to avoid seeking, while waiting for more sequential read requests to arrive. Anticipatory is a non-work conserving scheduler, because there are cases when it holds the system idle although there are outstanding requests waiting for service.
- **CFQ** (Command Fair Queuing) is also an elevator-based scheduler that, that in a multi-process environment, attempts to give every competing process the same number of IOs per unit of time in a round-robin fashion.

Figure 7 plots Postmark throughput for the four different workloads of Table 2 and the four different device driver scheduling algorithms. The file system is ReiserFS and the disk is the 18 GB one with a queue depth of 4. By fixing all system parameters, such as the file system, disk queue depth, and workload, we attribute the differences in Postmark throughput only to the device driver disk scheduling algorithm. Postmark throughput is clearly higher for scheduling algorithms that reorder the IO traffic based on inter-request seek distances. The differences are more noticeable under more random workloads (i.e., workloads SS and SL) than under more sequential workloads (i.e., workloads LS and LL).

No-Op is in clear disadvantage, with throughput as low as one fourth of the best performing scheduling algorithm (under the SS and LS workloads). As we explained earlier, this is an outcome of the non-work conserving nature of the disk scheduling algorithms where request reordering does result in less work to be done at the disk. In addition, No-Op performs poorly even for the more sequential LL workload, because it merges only consecutive sequential requests. Among the elevator-based disk schedulers, Deadline performs the best in our system, with a noticeable advantage only for the SS workload where randomness is high and working set size small.

Under all scenarios that we measured at the device driver level, the highest relative gain for a disk scheduling algorithm is approximately 20% once all other system parameters remain unchanged (this is for the case of ReiserFS, SS workload and Deadline and Anticipatory schedulers). This gain is much smaller than the case depicted in Figure 3 where the file system rather than the scheduling algorithm is the changing system parameter and the performance gains is as much as 80% for ReiserFS when compared with XFS under the SS workload.

Disk scheduling algorithms, including No-Op, merge incoming file system requests to exploit their temporal and spatial locality. Anticipatory scheduling is the most aggressive algorithm with respect to merging, because it waits up to 3 ms (in our configuration) for new read arrivals that could be merged with outstanding ones, even though the storage system might be idle. No-Op scheduling is the least efficient algorithm because it merges only consecutive requests that are sequentially located on the disk. Deadline and CFQ do not put conditions on consecutive arrivals of sequential requests as No-Op does.

Differences in disk scheduling algorithms with respect to request merging are depicted in Figure 8, where we plot the average request size for the read and write traffic in and out of the device driver for the four workloads of Table 2 under the four different disk scheduling algorithms. The file system is ReiserFS and the disk is the 18 GB one with a queue depth of 4. There is a substantial difference between No-Op write request size out of the device driver and other schedulers write request size out of the device driver. No-Op does not succeed to merge any write requests at all because their sequentiality is broken by other IO requests. In particular, under ReiserFS, which we plot in Figure 8, No-Op is under disadvantage because the file system chops application write requests to 4 KB. All other disk scheduling algorithms perform comparably, because the high IO

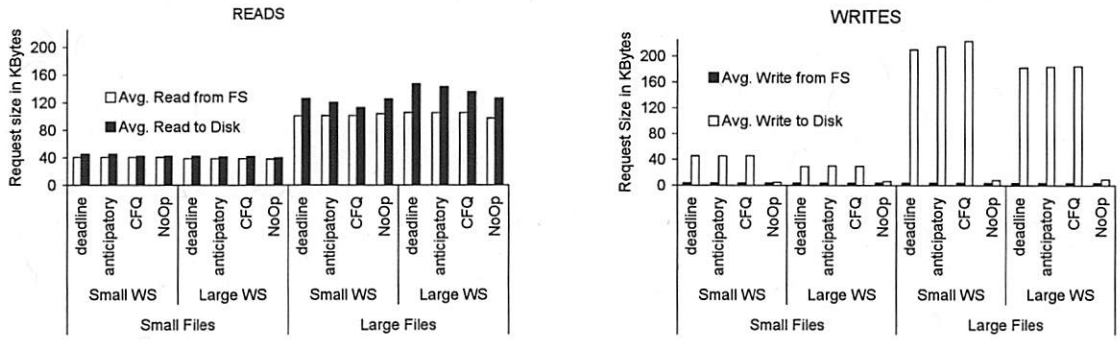


Figure 8: Average request size of reads (left) and writes (right) at the device driver for the incoming and outgoing traffic for the four workloads of Table 2 and four scheduling algorithms. Measurements are done on the 18 GB disk and ReiserFS.

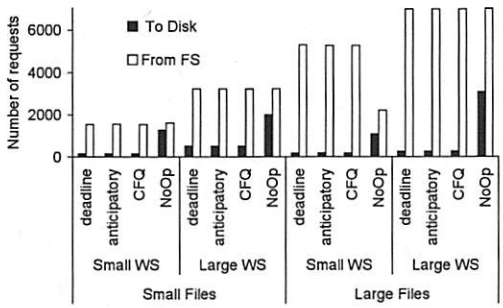
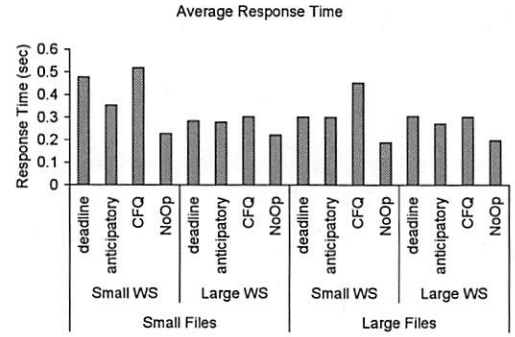


Figure 9: Number of requests in and out of the device driver level, for the four workloads of Table 2 and the four disk scheduling algorithms. Measurements are done on the 18 GB disk and ReiserFS.

load with short interarrival times causes these scheduling algorithms to meet their parameter thresholds for request merging. Generally, device driver disk scheduling merges more effectively writes than reads for the Postmark workloads evaluated here.

The same conclusion is drawn from the results presented in Figure 9, where we plot the number of requests in and out the device driver level for each scheduler and the four workloads of Table 2. For the workloads that access large files (i.e., LS and LL) the degree of merging is higher than for workloads that access small files (i.e., SS and SL), because the workload is sequential and, on its own, represents more opportunities for request merging.

In Figure 10, we show the average response time and queue length at the device driver level for the four workloads of Table 2, the four disk scheduling algorithms, ReiserFS, and the 18 GB disk with queue depth of 4.



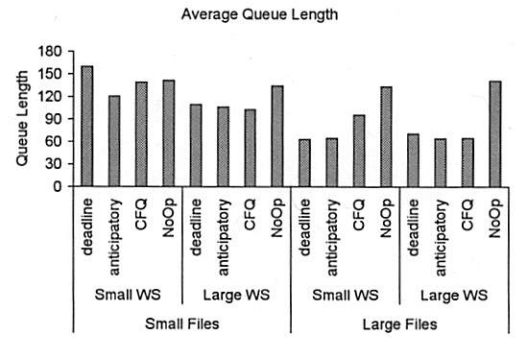


Figure 10: Average response time and queue length at the device driver for the four scheduling algorithms and the four workloads of Table 2. Measurements are done on the 18 GB disk and ReiserFS.

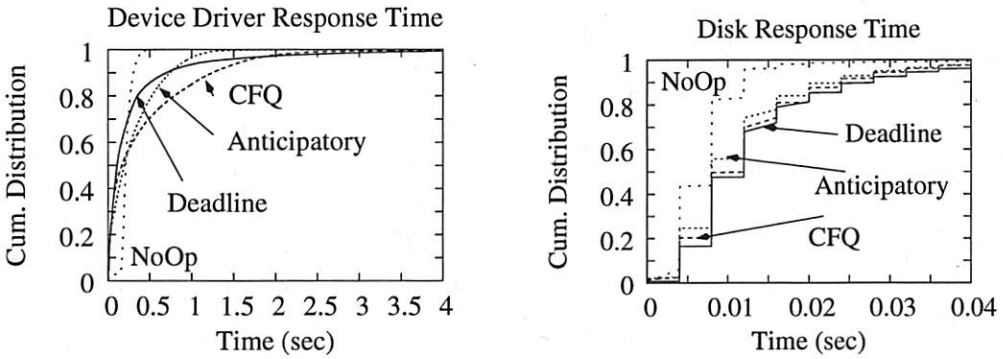


Figure 11: Response time distribution at the device driver and disk for the SS workload. Measurements are done on the 18 GB disk and ReiserFS.

The results of Figure 10 clearly indicate that the device driver operates under very high load with queue lengths in the hundred and response times in the level of seconds. Request response time under No-Op is better than under the other scheduling algorithms because the requests and, consequently, the disk transfer times are shorter under No-Op than under the other disk scheduling algorithms (see Figure 8). The difference is reflected in overall average request response time.

We also plot the cumulative distribution function of response times at the device driver and at the disk for only one workload (i.e., SS) and the four scheduling algorithms.in Figure 11 Indeed the disk scheduling algorithms other than No-Op do introduce more variability in the system as the distribution at the device driver indicates. However this variability does not affect the overall system performance since under these scheduling algorithms the application throughput is noticeably better than under the fair No-Op scheduler. The variability injected in the workload because of scheduling at the device driver level, does not get more pronounced at the disk level as the distribution of disk response time indicate. The difference between the four device driver schedulers in the disk response time distribution is attributed to the difference in request size between No-Op and the other three disk schedulers.

5.1 Second application: Linux kernel build

Previously, we showed that Postmark fully utilizes the storage subsystem (see Figure 10). As indicated in Section 4, under heavy load, request merging becomes the determining factor on IO performance and the best performer, i.e., ReiserFS, distinguishably sets itself apart from the other file systems. In order to evaluate behavior under lighter load conditions, we chose to generate IO workloads by compiling the Linux 2.6 kernel, twice and four times simultaneously. Note that a single Linux kernel compilation in our dual processor system performs the same as two simultaneous Linux kernel builds and more than four simultaneous builds load the system heavily (similarly to Postmark).

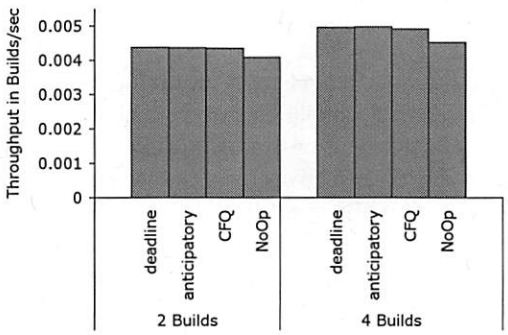


Figure 12: Throughput in builds per second for the Linux kernel compilation benchmark.

Linux kernel compilation generates a write-dominated random workload. Different builds were placed in different disk partitions which means that disk head positioning is involved when multiple simultaneous builds are running. Because the average request size is different for the Linux build measurements and the Postmark ones, we compare the load level by looking at the request interarrival times. While for Postmark the average interarrival time is 3 ms, for the Linux build is 19 ms and 12 ms for the two and four simultaneous builds.

Figure 12 shows the throughput of this benchmark measured in builds per second. Note that the differences between the disk scheduling algorithms at the device driver level are not as pronounced as in the case of the heavy Postmark load. Even No-Op is not far behind the seek-based scheduling algorithms. In the next section, we come back to the Linux kernel build bench-

mark and discuss the effectiveness of disk level request scheduling under medium IO load, where request merging has smaller effect on IO optimization than under the Postmark benchmark. Under the Linux build benchmark, request reordering is of more importance than request merging.

6 Disk Drive Level

The disk drive is the last component in the IO path that we analyze in this paper. With the advances in chip design, modern disks have more CPU and more memory available than before. Although various optimization techniques such as request merging are best suited for the upper levels of the IO path such as the file system and the device driver, the disk itself does offer various optimization opportunities, mostly related to caching of data and scheduling of requests. We stress that disk scheduling at the disk level is the most effective IO request scheduling in the IO path [21], because it uses information on head position that is available only at the disk drive level. In this section, we focus on evaluating disk level scheduling by analyzing the effectiveness of disk level queuing.

We use three different disks in our evaluation manufactured by Seagate Technology. The main differences between them are the linear density, rotational speed, and capacity, which determine the average head positioning time and the associated overhead for each disk request. Details on the three disks used in our evaluation are given in Table 3. Note that the linear density of the 300 GB disk is approximately 24% and 64% higher than the linear density of the 146 GB disk and the 18 GB disk, respectively, while its average seek time is higher by approximately 30%. Although seek time is expected to improve if disk tracks are closer together, in the case of the 300 GB drive, the platters are larger to accommodate the large capacity and consequently the arm itself becomes heavier, which results in longer seek times for the 300 GB disk than the 18 GB and 146 GB disks.

	ST318453LC	ST3146854LC	ST3300007LC
Capacity	18 GB	146 GB	300 GB
RPM	15,000	15,000	10,000
Platters	1	4	4
Linear density	64K TPI	85K TPI	105K TPI
Avg seek time	3.6/4 ms	3.4/4 ms	4.7/5.3 ms
Cache	8 MB	8 MB	8MB

Table 3: Specifications for the three Seagate disks.

In Figure 13, we show the Postmark throughput for the three disks of Table 3 and the SS and LL workloads of Table 2. The file system is changed while the disk scheduling is set to Anticipatory. The highest application throughput is achieved for the 146 GB disk, with the 18 GB to a close second. The higher average seek times and the slower rotation speed cause the 300 GB disk drive, although with the highest linear density, to achieve the lowest application throughput among the three disk drives.

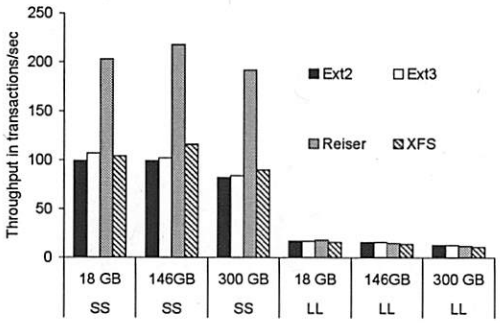


Figure 13: Postmark throughput for all disks of Table 3, two of the workloads of Table 2, the four file systems, anticipatory scheduling, and disk queue depth of 1.

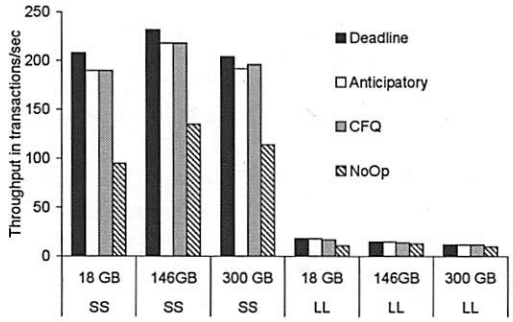


Figure 14: Postmark throughput for all disks of Table 3, two of the workloads of Table 2, the four disk scheduling algorithms, ReiserFS, and disk queue depth of 1.

Figure 14 is similar to Figure 13, but now the file system is fixed to ReiserFS and the Postmark performance is measured under four disk scheduling algorithms. As pointed out above, the 146 GB disk is the one performing best under all device driver disk scheduling algorithms. Note that the relative gap between No-Op and the other seek-based algorithms is smaller for the newer disks (i.e., 146 GB and 300 GB) than the older one (i.e., 18 GB).

6.1 Disk Level Queuing

As we mentioned previously, disk request scheduling is non-work conserving and the optimal algorithm, which is also NP-complete, uses the positioning time rather

than the seek time per request, when it comes to compute the optimal schedule [1]. Disk head positioning time is only accurately predicted at the disk drive itself, rather than any other level of the IO path. This is because the SCSI interface does not support sharing such information and because disks conduct various internal background activities to enhance their performance and reliability. Consequently, predicting disk head position at the device driver level is difficult, although, various efforts have been made to enhance disk scheduling at the device driver beyond the seek-based schedulers [5, 16].

Computing the optimal schedule is computationally expensive and impractical. Consequently, newer disks with more CPU and memory resources than before can easily accommodate long queue depths at the disk level and exploit almost all the performance enhancement available via disk level request reordering based on advanced heuristics rather than the optimal scheduling. The queuing buffer of the disks of Table 3 can hold up to 64 outstanding requests and they implement variants of the Shortest Positioning Time First algorithms [27].

The queue depth at the disk drive level is a parameter set at the device driver and often set to 4 (the default value for many SCSI device drivers). The queue depth at the disk is commonly kept low to avoid request starvation, because disk drive scheduling introduces variability in request response time, which can be controlled easily by the operating system at the device driver with algorithms such as Deadline but not at the disk drive level.

Here, we set the disk queue depth beyond the default 4 to evaluate the overall system benefits by queuing more at the disk drive level. To ease our presentation, we use only the 18 GB and the 300 GB disks from Table 3. The rest of the system configuration is: ReiserFS and the SS workload from Table 2. We use these settings because they represent the case with the most efficient and pronounced optimization (among all cases evaluated in the previous sections). We plot the measured Postmark throughput in Figure 15. The plot shows that, although the disk is the lowest component in the IO path, by only increasing the disk queue depth, we improve overall application performance and throughput.

The relative gain of deeper queues at the disk in Figure 15, is more pronounced for the newer 300 GB disk than the older 18 GB one. Actually for the best performing device driver scheduling algorithm (i.e., Deadline), the throughput of the 300 GB disk which we showed in Figures 14 and 13 to be lower than that of the other disks for the same workload, is very close to the throughput of the 18 GB disk.

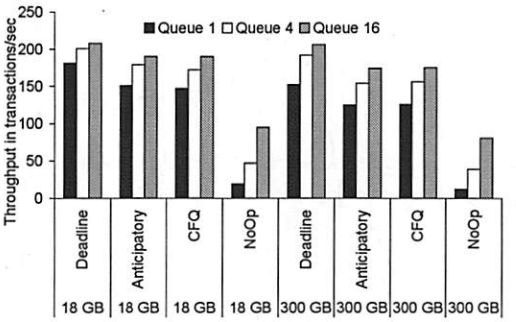


Figure 15: Postmark throughput as a function of the disk queue depth for the 18 GB and the 300 GB disks, ReiserFS, and the SS workload.

6.2 Disks with Write Cache Enabled

The cache at the disk, as anywhere else in the system, is an important resource that aids performance enhancement in the IO path. While evaluation of disk cache effectiveness is outside the scope of our analysis, in this section, we evaluate the relation between the effectiveness of disk level queuing and two different policies for using the available cache by the incoming write traffic, specifically, the "write-back" and the "write-through" polices.

The write-back policy is proposed to further enhance performance at the disk drive level. Under the write-back policy, once a write request has arrived at the disk cache, the host is notified for its completion. From the host perspective the write service time is equal to the disk cache hit time rather than the time to actually completely store the write on the disk media. The drawback of this policy is that it might lead to inconsistencies during system crashes or failures, because the data is not permanently stored while it appears so for the host system. Disk write-back policy is the default one for all the SATA drives, which are installed in systems with moderate reliability requirements. Interestingly enough, even the newer SCSI Seagate disks (i.e., the 146 GB and the 300 GB ones) came with the write cache enabled. It is true that if non-volatile memory is available in the IO path, then inconsistencies because of data loss during crashes are reduced. As such "write-back" is the default policy for disk drives.

The other option is to write the data through on the disk and notify the host that a write is completed only when the data is actually stored safely on the disk media. This means that read and write traffic are handled similarly and for almost all writes the disk service time is not going to be the disk cache-hit time anymore. This policy is called "write-though" and provides high levels of

data reliability and consistency. Disk level performance under the "write-through" policy is worse than under the "write-back" policy, which explains why the former is used only in systems with the highest level of data reliability and consistency requirements. All our previous results in this paper are generated with "write-through" disk caches.

In this subsection, we evaluate the system and application behavior when the only parameter that changes in our testbed is the disk queue depth while we set the file system to be ReiserFS and run the SS workload. We also set the disk cache to be "write-back". We conduct measurements for all four device driver schedulers in our evaluation.

We present Postmark throughput in Figure 16. Without disk-level queuing (i.e., queue depth of one) Postmark achieves the highest (or very close to it) throughput possible for the configuration. Additional queuing at the disk, commonly results in performance degradation. In particular, this is the case when the device driver scheduler, such as No-Op, is not effective on IO request ordering and merging.

The adversary affect of disk level queuing on overall system performance when "write-back" is enabled is related to the limited queueing buffer space at the disk. If write cache is enabled, the effective disk queue is longer than what disk advertises at the device driver. Hence, under heavy loads, as it is the case for Postmark, the actual disk queue reaches the physical maximum allowable queue of 64 for the disks under our evaluation. If the buffer queue at the disk is full, the disk responds to the host with a "disk full" message, which indicates that it can not accept more requests. Consequently, the device driver delays the requests longer and because the IO subsystem does operate as a closed system, these delays propagate and affect overall system performance, i.e., the system slows down. As it can be seen from Figure 16, the negative disk queuing effect is not consistent. Specifically, the minimum throughput is often at queue depth of 4 and not at the higher queue depth of 16.

Comparing results in Figure 15 with results in Figure 16, we observe that if the disk "write-through" policy is enabled as well as the queue depth is set to a high value, then highest Postmark throughput achieved under "write-through" and "write-back" policies are very close. This means that the level of optimization offered in the IO path, in particular disk level queuing, is as effective as to close the performance gap between the disk "write-through" and "write-back" policies. We conclude this subsection, by stressing that deep disk queues and "write-through" policy not only achieve application-level throughput as high as under the performance-enhancer "write-back" policy, but enhance the always needed data consistency and reliability.

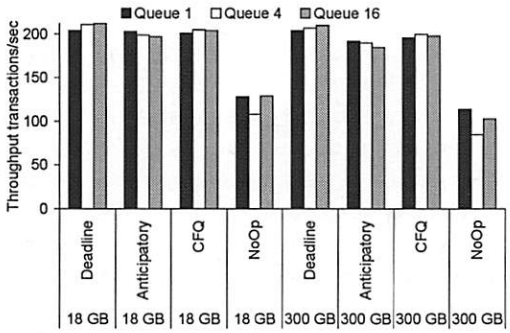


Figure 16: Postmark throughput as a function of the disk queue depth for the 18 GB and 300 GB disks, ReiserFS and the "small files, small working set" (SS) workload. Write-back cache is enabled at the disk.

6.3 Second application: Linux kernel build

As we discussed in more length in Subsection 5.1, Postmark load is high and we choose to benchmark also a case of medium load in the system by compiling simultaneously two and four Linux kernels. Under high load, as it is the case of Postmark, IO requests wait mostly at the device driver, because the number of outstanding IO requests is higher than the maximum disk queue depth. Hence disk level queuing is effective only for a small portion of the outstanding IO requests and optimization effectiveness at the device driver becomes more important than queuing at the disk drive level.

Under medium loads, as it is the case of the Linux kernel compilation, which we described in Subsection 5.1, the impact of disk queuing in the overall system performance is higher than under high loads. This holds in particular for cases when the optimization at the device driver level (or any other level in the IO path) is not as effective as it could be (as it is the case of the No-Op scheduler).

In Figure 17, we show the effectiveness of disk queuing when compiling 2 and 4 Linux kernels simultaneously. Note that No-Op benefits the most from disk queuing and the result is that the overall system performance (measured in builds per seconds) is very close for all four device driver disk schedulers when disk queue depth is 16 or higher. Hence, we conclude that disk level queuing and scheduling is effective and does close any performance gaps in the IO request reordering from the higher levels in the IO path.

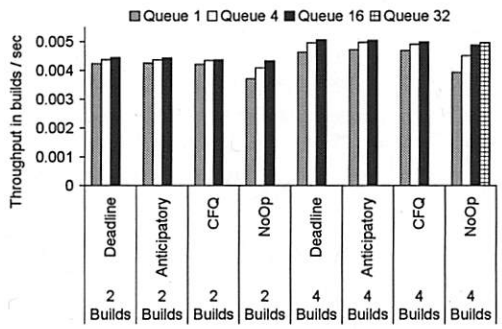


Figure 17: Throughput in builds per second for the four disk scheduling algorithms, and different disk queue depths. Measurements are done on the 18 GB disk and ReiserFS.

7 Related Work

Because the performance gap between memory and disk drives has remained significant, even with the latest advances in technology, optimizing performance in the IO path has been the focus of work for many researchers. In addition to adding resources such as caches [2] in the IO path and managing them more efficiently [14], optimization of the IO activity itself is one of the main tools to enhance IO performance. IO workload optimization consists mostly of request merging and request reordering at various levels of the IO hierarchy, such as the file system, the device driver, and the disk drive.

IO work reordering via scheduling early on aimed at minimizing the linear disk head movement (i.e., seeks) [3, 4, 7, 20] and later evolved to minimizing the overall head position phase [1, 10, 21] of a disk request service. Because IO request reordering introduces variability in request response time, multiple variations are proposed to mitigate the problem [27]. On the analytic side, there are studies that compare analytically disk scheduling algorithms [25] and derive empirical models of disk access patterns to facilitate such comparison [26].

Recent advances in disk scheduling include speculative waiting to better exploit temporal and spatial locality of IO requests [9], predicting disk service times [16] and disk head position [5] to enhance performance of seek-based schedulers at the device driver, increasing disk head utilization by reading "for free" data on the path of the disk head movement [13], and other hierarchical approaches [22], which are mostly used for performance virtualization in large systems [12].

Performance optimization at the disk drive level is mostly related to request scheduling and evaluated in association with it [27]. IO performance improvement is also evaluated in association with workload characterization studies [19]. Performance enhancement related to advancements in disk technology, including areal density and rotational speed is discussed in [15]..

While most request reordering happens either at the device driver or at the disk, file systems also represent a critical component of the IO path capable of effective IO work optimization. The main research efforts when it comes to enhancing file systems, are related with improving data reliability and consistency [17, 23]. However considerable work is done to enhance file system performance as well. For example scalability of the XFS file system is discussed in [24] and file system workload characteristics are analyzed in [6, 8]. A comparative study between file systems is presented in [18].

Apart from the existing work, our paper analyzes advances in the IO path, and evaluates how effectively they integrally optimize and enhance IO performance. Most works evaluate the components of the IO path individually, while in this paper, we analyze their impact in overall application performance. Our goal is to first identify the reasons why some components in the IO path are more effective than others in optimizing the IO workload and secondly understand if and how such optimization can be as effective in other tiers as well with the goal to further enhance IO subsystem performance.

8 Conclusions

In this paper, we presented a measurement-based analysis of the optimization effectiveness in the IO subsystem, focusing on the file system, device driver scheduler, and the disk itself. We used the Postmark benchmark to generate heavy IO write-intensive workload and the Linux kernel compilation to generate medium IO write-intensive workload. We analyzed four file systems, four device driver level disk schedulers and three Seagate disk drives.

Our measurements showed that request merging is critical for reducing the number of disk requests and enhancing overall performance. Although, request merging takes place at the device driver scheduler, it is the block allocation and data management at the file system itself that determines the effectiveness of the request merging process. The most effective file system in our measurements was ReiserFS which improved by as high as 100% application-level throughput.

Under medium load though, request merging becomes less efficient and request reordering becomes the tool to optimize the IO traffic. Request reordering at the device driver level improved application-level throughput by as much as 20% application-level throughput under the Deadline device driver scheduler.

Disk drives have become very effective on optimizing request reordering, closing any performance gaps between the elevator-based and FCFS device driver schedulers. Increasing the queue depth at the disk drive under the write-through cache policy, improves the overall application throughput with as much as 30% if the device driver scheduler is seek-based and more than 6 times if the device driver scheduler is FCFS (No-Op). On the contrary, under heavy load, disk queuing has a negative impact on application throughput when the write-back cache policy is enabled. Overall, combining the write-through disk cache policy and high queue depths performs similarly with the write-back disk cache policy and it is attractive because it does not compromise data reliability and consistency, as the write-back cache policy does.

References

[1] ANDREWS, M., BENDER, M. A., AND ZHANG, L. New algorithms for the disk scheduling problem. *Algorithmica 32*, 2 (2002), 277–301.

[2] BAKER, M., ASAMI, S., DEPRIT, E., OUSTERHOUT, J. K., AND SELTZER, M. I. Non-volatile memory for fast, reliable file systems. In *ASPLOS* (1992), pp. 10–22.

[3] COFFMAN, E. G., AND HOFRI, M. On the expected performance of scanning disks. *SIAM Journal of Computing 10*, 1 (1982), 60–70.

[4] DENNING, P. J. Effects of scheduling on file memory operations. In *Proceedings of AFIPS Spring Joint Computer Conference* (1967), pp. 9–21.

[5] DIMITRIJEVIC, Z., RANGASWAMI, R., AND CHANG, E. Y. Systems support for preemptive disk scheduling. *IEEE Trans. Computers 54*, 10 (2005), 1314–1326.

[6] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. Passive nfs tracing of email and research workloads. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), USENIX Association, pp. 203–216.

[7] GEIST, R., AND DANIEL, S. A continuum of disk scheduling algorithms. *ACM transactions on Computer systems 5*, 1 (1987), 77–92.

[8] GRIBBLE, S. D., MANKU, G. S., ROSELLI, D., BREWER, E. A., GIBSON, T. J., AND MILLER, E. L. Self-similarity in file systems. In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems* (1998), ACM Press, pp. 141–150.

[9] IYER, S., AND DRUSCHEL, P. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles* (Oct. 2001).

[10] JACOBSON, D. M., AND WILKES, J. Disk scheduling algorithms based on rotational position. Tech. Rep. HPL-CSP-91-7rev1, HP Laboratories, 1991.

[11] KATCHER, J. Postmark: A new file system benchmark. Tech. Rep. 3022, Network Appliances, Oct. 1997.

[12] LUMB, C. R., MERCHANT, A., AND ALVAREZ, G. A. Façade: Virtual storage devices with performance guarantees. In *FAST* (2003).

[13] LUMB, C. R., SCHINDLER, J., GANGER, G. R., NAGLE, D., AND RIEDEL, E. Towards higher disk head utilization: Extracting "free" bandwidth from busy disk drives. In *OSDI* (2000), pp. 87–102.

[14] MEGIDDO, N., AND MODHA, D. S. Arc: A self-tuning, low overhead replacement cache. In *FAST* (2003).

[15] NG, S. W. Advances in disk technology: Performance issues. *IEEE Computer 31*, 5 (1998), 75–81.

[16] POPOVICI, F. I., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Robust, portable i/o scheduling with the disk mimic. In *Proceedings of Annual USENIX Technical Conference* (San Antonio, TX, June 2003), pp. 311–324.

[17] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference, General Track* (2005), pp. 105–120.

[18] ROSELLI, D., LORCH, J. R., AND E.ANDERSON, T. A comparison of file systems workloads. In *Proceedings of USENIX Technical Annual Conference* (2000), pp. 41–54.

[19] RUEMMLER, C., AND WILKES, J. Unix disk access patterns. In *Proceedings of the Winter 1993 USENIX Technical Conference* (1993), pp. 313–323.

[20] SEAMAN, P. H., LIND, R. A., AND WILSON, T. L. An analysis of auxilary-storage activity. *IBM System Journal 5*, 3 (1966), 158–170.

[21] SELTZER, M., CHEN, P., AND OSTERHOUT, J. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX Technical Conference* (Washington, DC, 1990), pp. 313–323.

[22] SHENOY, P., AND VIN, H. M. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of ACM SIGMETRICS Conference, Madison, WI* (June 1998), pp. 44–55.

[23] STEIN, C. A., HOWARD, J. H., AND SELTZER, M. I. Unifying file system protection. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 79–90.

[24] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Technical Conference* (San Diego, CA, USA, 22–26 1996), pp. 1–14.

[25] TEOREY, T. J., AND PINKERTON, T. B. A comparative analysis of disk scheduling policies. *Commun. ACM 15*, 3 (1972), 177–184.

[26] WILHELM, N. C. An anomaly in disk scheduling: a comparison of fcfs and sstf seek scheduling using an empirical model for disk accesses. *Commun. ACM 19*, 1 (1976), 13–17.

[27] WORTHINGTON, B. L., GANGER, G. R., AND PATT, Y. N. Scheduling algorithms for modern disk drives. *SIGMETRICS Perform. Eval. Rev. 22*, 1 (1994), 241–251.

DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch

Xiaoning Ding[1], Song Jiang[2], Feng Chen[1], Kei Davis[3], and Xiaodong Zhang[1]

[1] *CSE Department*
Ohio State University
Columbus, OH 43210, USA
{dingxn,fchen,zhang}@cse.ohio-state.edu

[2] *ECE Department*
Wayne State University
Detroit, MI 48202, USA
sjiang@eng.wayne.edu

[3] *CCS-3 Division*
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
kei.davis@lanl.gov

Abstract

Current disk prefetch policies in major operating systems track access patterns at the level of the file abstraction. While this is useful for exploiting application-level access patterns, file-level prefetching cannot realize the full performance improvements achievable by prefetching. There are two reasons for this. First, certain prefetch opportunities can only be detected by knowing the data layout on disk, such as the contiguous layout of file metadata or data from multiple files. Second, non-sequential access of disk data (requiring disk head movement) is much slower than sequential access, and the penalty for mis-prefetching a 'random' block, relative to that of a sequential block, is correspondingly more costly.

To overcome the inherent limitations of prefetching at the logical file level, we propose to perform prefetching directly at the level of disk layout, and in a portable way. Our technique, called *DiskSeen*, is intended to be supplementary to, and to work synergistically with, file-level prefetch policies, if present. *DiskSeen* tracks the locations and access times of disk blocks, and based on analysis of their temporal and spatial relationships, seeks to improve the sequentiality of disk accesses and overall prefetching performance.

Our implementation of the DiskSeen scheme in the Linux 2.6 kernel shows that it can significantly improve the effectiveness of prefetching, reducing execution times by 20%-53% for micro-benchmarks and real applications such as *grep*, *CVS*, and *TPC-H*.

1 Introduction

As the speed differential between processor and disk continues to widen, the effect of disk performance on the performance of data-intensive applications is increasingly great. Prefetching—speculative reading from disk based on some prediction of future requests—is a fundamental technique for improving effective disk performance. Prefetch policies attempt to predict, based on analysis of disk requests, the optimal stream of blocks to prefetch to minimize disk service time as seen by the application workload. Prefetching improves disk performance by accurately predicting disk requests in advance of the actual requests and exploiting hardware concurrency to hide disk access time behind useful computation.

Two factors demand that prefetch policies be concerned with not just accuracy of prediction, but also actual time cost of individual accesses. First, a hard disk is a non-uniform-access device for which accessing sequential positions without disk head movement is at least an order of magnitude faster than random access. Second, an important observation is that as an application load becomes increasingly I/O bound, such that disk accesses can be decreasingly hidden behind computation, the importance of sequential prefetching increases relative to the importance of prefetching random (randomly located) blocks. This is a consequence of the speculative nature of prefetching and the relative penalties for incorrectly prefetching a sequential block versus a random block. This may explain why, despite considerable work on sophisticated prefetch algorithms (Section 5), general-purpose operating systems still provide only sequential prefetching or straightforward variants thereof. Another possible reason is that other proposed schemes have been deemed either too difficult to implement relative to their probable benefits, or too likely to hurt performance in some common scenarios. To be more relevant to current practice, the following discussion is specific to prefetch policies used in general-purpose operating systems.

Existing prefetch policies usually detect access patterns and issue prefetch requests at the logical file level. This fits with the fact that applications make I/O requests based on logical file structure, so their discernible access patterns will be directly in terms of logical file structure. However, because disk data layout information is not exploited by these policies, they do not have the knowledge of where the next prefetched block would be relative to the currently fetched block to estimate prefetching cost. Thus, their measure of prefetching effectiveness, which is usually used as a feedback to adjust prefetching behavior, is in terms of the number of mis-prefetched blocks rather than a more relevant metric, the penalty of mis-prefetching. Disk layout information is not used until the requests are processed by the lower-level disk scheduler where requests are sorted and merged, based on disk placement, into a dispatching queue using al-

gorithms such as SSTF or C-SCAN to maximize disk throughput.

We contend that file-level prefetching has both practical and inherent limitations, and that I/O performance can be significantly improved by prefetching based on disk data layout information. This disk-level prefetching is intended to be supplementary to, and synergistic with, any file-level prefetching. Following we summarize the limitations of file-level prefetching.

Sequentiality at the file abstraction may not translate to sequentiality on disk. While file systems typically seek to dynamically maintain a correspondence between logical file sequentiality and disk sequentiality, as the file system ages (e.g. in the case of Microsoft's NTFS) or becomes full (e.g. Linux Ext2) this correspondence may deteriorate. This worsens the penalty for mis-prediction.

The file abstraction is not a convenient level for recording deep access history information. This is exacerbated by the issue of maintaining history information across file closing and re-opening and other operations by the operating system. As a consequence, current prefetch schemes maintain shallow history information and so must prefetch conservatively [21].[1] A further consequence is that sequential access of a short file will not trigger the prefetch mechanism.

Inter-file sequentiality is not exploited. In a general-purpose OS, file-level prefetching usually takes place within individual files, which precludes detection of sequential access across contiguous files.

Finally, blocks containing file system metadata cannot be prefetched. Metadata blocks, such as inodes, are not in files, and so cannot be prefetched. Metadata blocks may need to be visited frequently when a large number of small files are accessed.

In response, we propose a disk-level prefetching scheme, *DiskSeen*, in which current and historical information is used to achieve *efficient* and *accurate* prefetching. While caches in hard drives are used for prefetching blocks directly ahead of the block being requested, this prefetching is usually carried out on each individual track and does not take into account the relatively long-term temporal and spatial locality of blocks across the entire disk working set. The performance potential of the disk's prefetching is further constrained because it cannot communicate with the operating system to determine which blocks are already cached there; this is intrinsic to the disk interface. The performance improvements we demonstrate are in addition to those provided by existing file-level and disk-level prefetching.

We first describe an efficient method for tracking disk block accesses and analyzing associations between blocks (Section 2). We then show how to efficiently detect sequences of accesses of disk blocks and to appropriately initiate prefetching at the disk level. Further aided by access history information, we show how to detect complicated pseudo-sequences with high accuracy (Section 3). We show that an implementation of these algorithms—collectively DiskSeen—in the current Linux kernel can yield significant performance improvements on representative applications (Section 4).

2 Tracking Disk Accesses

There are two questions to answer before describing DiskSeen. The first is what information about disk locations and access times should be used by the prefetch policy. Because the disk-specific information is exposed using the unit of disk blocks, the second question is how to efficiently manage the potentially large amount of information. In this section, we answer these two questions.

2.1 Exposing Disk Layout Information

Generally, the more specific the information available for a particular disk, the more accurate an estimate a disk-aware policy can make about access costs. For example, knowing that blocks span a track boundary informs that access would incur the track crossing penalty [24]. As another example, knowing that a set of non-contiguous blocks have some spatial locality, the scheduler could infer that access of these blocks would incur the cost of semi-sequential access, intermediate between sequential and random access [26]. However, detailed disk performance characterization requires knowledge of physical disk geometry, which is not disclosed by disk manufacturers, and its extraction, either interrogative or empirical, is a challenging task [30, 23]. Different extraction approaches may have different accuracy and work only with certain types of disk drives (such as SCSI disks).

An interface abstraction that disk devices commonly provide is logical disk geometry, which is a linearized data layout and represented by a sequence $[0, 1, 2, ..., n]$ of *logical block numbers* (LBNs). Disk manufacturers usually make every effort to ensure that accessing blocks with consecutive LBNs has performance close to that of accessing contiguous blocks on disk by carefully mapping logical blocks to physical locations with minimal disk head positioning cost [26]. Though the LBN does not disclose precise disk-specific information, we use it to represent disk layout for designing a disk-level prefetch policy because of its standardized availability and portability across various computing platforms. In this paper, we will show that *exposing*[2] this logical disk layout is sufficient to demonstrate that incorporating disk-side information with application-side information into prefetch policies can yield significant performance benefits worthy of implementation.

2.2 The Block Table for Managing LBNs

Currently LBNs are only used to identify locations of disk blocks for transfer between memory and disk. Here we track the access times of recently touched disk blocks via their LBNs and analyze the associations of access times among adjacent LBNs. The data structure holding this information must support efficient access of block entries and their neighboring blocks via LBNs, and efficient addition and removal of block entries.

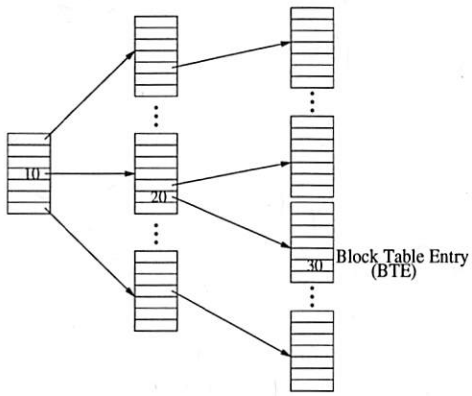


Figure 1: **Block table**. There are three levels in the example block table: two directory levels and one leaf level. The table entries at differing levels are fit into separate memory pages. An entry at the leaf level is called a block table entry (BTE). If one page can hold 512 entries, the access time of a block with LBN 2,631,710 ($10\times512^2+20\times512+30$) is recorded at the BTE entry labeled 30, which can be efficiently reached via directory-level entries labeled 10 and 20.

The block table, which has been used in the DULO scheme for identifying block sequences [12], is inspired by the multi-level page table used for a process's memory address translation, which is used in almost all operating systems. As shown in Figure 1, an LBN is broken into multiple segments, each of which is used as an offset in the corresponding level of the table. In the DULO scheme, bank clock time, or block sequencing time, is recorded at the leaf level (i.e., block table entry (BTE)) to approximate block access time. In DiskSeen, a finer block access timing mechanism is used. We refer to the entire sequence of accessed disk blocks as the *block access stream*. The nth block in the stream has *access index* n. In DiskSeen, an access counter is incremented with each block reference; its value is the access index for that block and is recorded in the corresponding block table entry to represent the access time.

To facilitate efficient removal of old BTEs, each directory entry records the largest access index of all of the blocks under that entry. Purging the table of old blocks involves removing all blocks with access indices smaller than some given index. The execution of this operation entails traversing the table, top level first, identifying access indices smaller than the given index, removing the corresponding subtrees, and reclaiming the memory.

3 The Design of *DiskSeen*

In essence, DiskSeen is a sequence-based history-aware prefetch scheme. We leave file-level prefetching enabled; DiskSeen concurrently performs prefetching at a lower level to mitigate the inadequacies of file-level prefetching. DiskSeen seeks to detect sequences of block accesses based on LBN. At the same time, it maintains block access history and uses the history information to further improve the effectiveness of prefetching when recorded access patterns are observed to be repeated. There are four objectives in the design of DiskSeen.

1. *Efficiency.* We ensure that prefetched blocks are in a localized disk area and are accessed in the ascending order of their LBNs for optimal disk performance.
2. *Eagerness.* Prefetching is initiated immediately when a prefetching opportunity emerges.
3. *Accuracy.* Only the blocks that are highly likely to be requested are prefetched.
4. *Aggressiveness.* Prefetching is made more aggressive if it helps to reduce request service times.

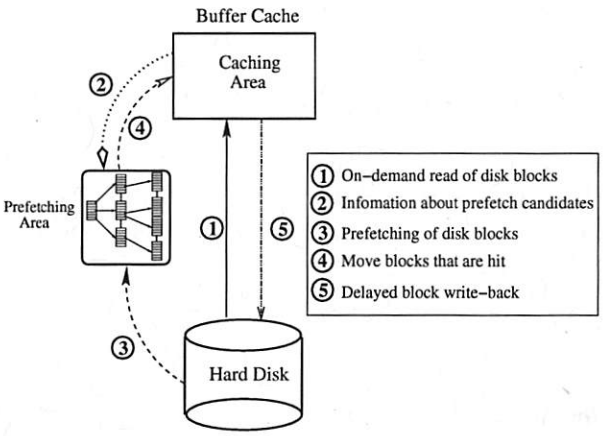


Figure 2: **DiskSeen system diagram**. Buffer cache is divided into two areas, prefetching and caching areas, according to their roles in the scheme. A block could be prefetched into the prefetching area based on either current or historical access information—both are recorded in the disk block table, or as directed by file-level prefetching. The caching area corresponds to the traditional buffer cache and is managed by the existing OS kernel policies except that prefetched but not-yet-requested blocks are no longer stored in the cache. A block is read into the caching area either from the prefetching area, if it is hit there, or directly from disk, all in an on-demand fashion.

As shown in Figure 2, the buffer cache managed by DiskSeen consists of two areas: prefetching and caching areas. The caching area is managed by the existing OS kernel policies, to which we make little change for the sake of generality. We do, however, reduce the size of

the caching area and use that space for the prefetching area to make the performance comparison fair.

DiskSeen distinguishes on-demand requests from file-level prefetch requests, basing disk-level prefetch decisions only on on-demand requests, which reflect applications' actual access patterns. While DiskSeen generally respects the decisions made by a file-level prefetcher, it also attempts to identify and screen out inaccurate predictions by the prefetcher using its knowledge of deep access history. To this end, we treat the blocks contained in file-level prefetch requests as prefetch candidates and pass them to DiskSeen, rather than passing the requests directly to disk. DiskSeen forwards on-demand requests from existing request mechanisms directly to disk. We refer to disk requests from 'above' DiskSeen (e.g., application or file-level prefetchers) as *high-level requests*.

3.1 Recording Access Indices

Block access indices are read from a counter that increments whenever a block is transferred into the caching area on demand. When the servicing of a block request is completed, either via a hit in the prefetching area or via the completion of a disk access, the current reading of the counter, an access index, is used as an access time to be recorded in the corresponding BTE in the block table. Each BTE holds the most recent access indices, to a maximum of four. In our prototype implementation, the size of a BTE is 128 bits. Each access index takes 31 bits and the other 4 bits are used to indicate block status information such as whether a block is resident in memory. With a block size of 4K Bytes, the 31-bit access index can distinguish accesses to 8 TBytes of disk data. When the counter approaches its maximum value, specifically the range for used access index exceeds 7/8 of the maximum index range, we remove the indices whose values are in the first half of the used range in the block table. In practice this progressive *index clearing* takes place very infrequently and its impact is minimal. In addition, a block table that consumes 4MB of memory can record history for about 1GB file access working set.

3.2 Coordinating Disk Accesses

We monitor the effectiveness of high-level prefetchers by tracking the use of prefetch candidates by applications. When a prefetch candidate block is read into the prefetching area, we mark the status of the block as *prefetched* in its BTE. This status can only be removed when an on-demand access of the block occurs. When the high-level prefetcher requests a prefetch candidate that is not yet resident in memory and has the *prefetched* status, DiskSeen ignores this candidate. This is because a previous prefetching of the block has not been followed by any on-demand request for it, which suggests an inaccurate prediction on the block made by the high-level prefetcher. This ability to track history prefetching events allows DiskSeen to identify and correct some of the misprefetchings generated by file-level prefetch policies.

For some access patterns, especially sequential accesses, the set of blocks prefetched by a disk-level prefetcher may also be prefetch candidates of file-level prefetchers or may be on-demand requested by applications. So we need to handle potentially concurrent requests for the same block. We coordinate these requests in the following way. Before a request is sent to the disk scheduler to be serviced by disk, we check the block(s) contained in the request against corresponding BTEs to determine whether the blocks are already in the prefetching area. For this purpose, we designate a *resident* bit in each BTE, which is set to 1 when a block enters buffer cache, and is reset to 0 when it leaves the cache. There is also a *busy* bit in each BTE that serves as a lock to coordinate simultaneous requests for a particular block. A set busy bit indicates that a disk service on the corresponding block is under way, and succeeding requests for the block must wait on the lock. DiskSeen ignores prefetch candidates whose resident or busy bits are set.

3.3 Sequence-based Prefetching

The access of each block from a high-level request is recorded in the block table. Unlike maintaining access state per file, per process, in file-level prefetching, DiskSeen treats the disk as a one-dimensional block array that is represented by leaf-level entries in the block table. Its method of sequence detection and access prediction is similar in principle to that used for the file-level prefetchers in some popular operating systems such as Linux and FreeBSD [2, 20].

3.3.1 Sequence Detection

Prefetching is activated when accesses of K contiguous blocks are detected, where K is chosen to be 8 to heighten confidence of sequentiality. Detection is carried out in the block table. For a block in a high-level request we examine the most recent access indices of blocks physically preceding the block to see whether it is the Kth block in a sequence. This back-tracking operation on the block table is an efficient operation compared to disk service time. Because access of a sequence can be interleaved with accesses in other disk regions, the most recent access indices of the blocks in the sequence are not necessarily consecutive. We only require that access indices of the blocks be monotonically decreasing. However, too large a gap between the access indices of two contiguous blocks indicates that one of the two blocks might not be accessed before being evicted from the prefetching area (i.e., from memory) if they

were prefetched together as a sequence. Thus these two blocks should not be included in the same sequence. We set an access index gap threshold, T, as $1/64$ of the size of the total system memory, measured in blocks.

3.3.2 Sequence-based Prefetching

When a sequence is detected we create two 8-block windows, called the current window and the readahead window. We prefetch 8 blocks immediately ahead of the sequence into the current window, and the following 8 blocks into the readahead window. We then monitor the number f of blocks that are hit in the current window by high-level requests. When the blocks in the readahead window start to be requested, we create a new readahead window whose size is $2f$, and the existing readahead window becomes the new current window, up to a maximum window size. Specifically, we set minimal and maximum window sizes, *min* and *max*, respectively. If $2f < min$, the prefetching is canceled. This is because requesting a small number of blocks cannot amortize a disk head repositioning cost and so is inefficient. If $2f > max$, the prefetching size is max. This is because prefetching too aggressively imposes a high risk of mis-prefetching and increases pressure on the prefetching area. In our prototype, min is 8 blocks and max is 32 blocks (with block size of 4KB). We note that the actual number of blocks that are read into memory can be less than the prefetch size just specified because resident blocks in the prefetch scope are excluded from prefetching. That is, the window size becomes smaller when more blocks in the prefetch scope are resident. Accordingly, prefetching is slowed down, or even stopped, when many blocks to be prefetched are already in memory.

3.3.3 Data Structure for Managing Prefetched Blocks

In the DiskSeen scheme, each on-going prefetch is represented using a data structure called the prefetch stream. The prefetch stream is a pseudo-FIFO queue where prefetched blocks in the two windows are placed in the order of their LBNs. A block in the stream that is hit moves immediately to the caching area. For one or multiple running programs concurrently accessing different disk regions, there would exist multiple streams. To facilitate the replacement of blocks in the prefetching area, we have a global FIFO queue called the reclamation queue. All prefetched blocks are placed at the queue tail in the order of their arrival. Thus, blocks in the prefetch windows appear in both prefetch streams and the reclamation queue.[3] A block leaves the queue either because it is hit by a high-level request or it reaches the head of the queue. In the former case the block enters the caching area, in the latter case it is evicted from memory.

3.4 History-aware Prefetching

In the sequence-based prefetching, we only use the block accesses of current requests, or recently detected access sequences, to initiate sequential prefetching. Much richer history access information is available in the block table, which can be used to further improve prefetching.

3.4.1 Access Trails

To describe access history, we introduce the term *trail* to describe a sequence of blocks that have been accessed with a small time gap between each pair of adjacent blocks in the sequence and are located in a pre-determined region. Suppose blocks $(B_1, B_2, ..., B_n)$ are a trail, where $0 < access_index(B_i) - access_index(B_{i-1}) < T$, and $|LBN(B_i) - LBN(B_1)| < S$, $(i = 2, 3, ..., n)$, where T is the same access index gap threshold as the one used in the sequence detection for the sequence-based prefetching. A block can have up to four access indices, any one of which can be used to satisfy the given condition. If B_1 is the start block of the trail, all of the following blocks must be on either side of B_1 within distance S. We refer to the window of $2S$ blocks, centered at the start block, as the *trail extent*. The sequence detected in sequence-based prefetching is a special trail in which all blocks are on the same side of start block and have contiguous LBNs. By using a window of limited size (in our implementation S is 128), we allow a trail to capture only localized accesses so that prefetching such a trail is efficient and the penalty for a mis-prefetching is small. For an access pattern with accesses over a large area, multiple trails would be formed to track each set of proximate accesses rather than forming an extended trail that could lead to expensive disk head movements. Trail detection is of low cost because, when the access index of one block in a trail is specified, at most one access index of its following block is likely to be within T. This is because the gap between two consecutive access indices of a block is usually very large (because they represent access, eviction, and re-access). Figure 3 illustrates.

3.4.2 Matching Trails

While the sequence-based prefetching only relies on the current on-going trail to detect a pure sequence for activating prefetching, we now can take advantage of history information, if available, to carry out prefetching even if a pure sequence cannot be detected, or to prefetch more accurately and at the right time. *The general idea is to use the current trail to match history trails and then use matched history trails to identify prefetchable blocks.* Note that history trails are detected in real-time and that there is no need to explicitly record them.

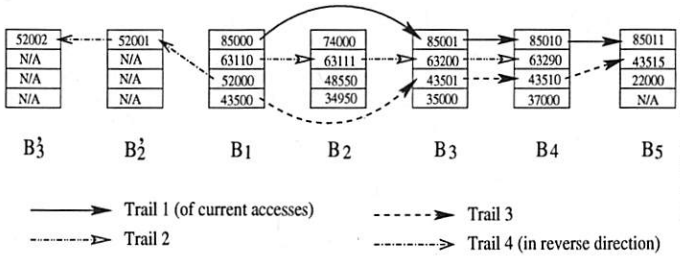


Figure 3: **Access trails.** Access index threshold T is assumed to be 256. There are four trails starting from block B_1 in a segment of the block table: one *current trail* and three *history trails*. Trail 1 (B_1, B_3, B_4, B_5) corresponds to the on-going continuous block accesses. This trail cannot lead to a sequence-based prefetch because B_2 is missing. It is echoed by two history trails: Trails 2 and 3, though Trail 1 only overlaps with part of Trail 2. A trail may run in the reverse direction, such as Trail 4.

When there is an on-demand access of a disk block that is not in any current trail's extent, we start tracking a new trail from that block. Meanwhile, we identify history trails consisting of blocks visited by the current trail in the same order. Referring to Figure 3, when the current trail extends from $B_1(85000)$ to $B_3(85001)$, two history trails are identified: Trail 2 ($B_1(63110)$, $B_3(63200)$) and Trail 3 ($B_1(43500)$, $B_3(43501)$). When the current trail advances to block B_4, both Trail 2 and Trail 3 successfully extend to it. However, only Trail 3 can match the current trail to B_5 while Trail 2 is broken at the block.

3.4.3 History-aware Prefetching

Because of the strict matching requirement, we initiate history-aware prefetching right after we find a history trail that matches the current trail for a small number of blocks (4 blocks in the prototype). To use the matched history trails to find prefetchable blocks, we set up a trail extent centered at the last matched block, say block B. Then we run the history trails from B in the extent to obtain a set of blocks that the matched history trails will probably visit. Suppose ts is an access index of block B that is used in forming a matched history trail, and T is access index gap threshold. We then search the extent for the blocks that contain an access index between ts and $ts + T$. We obtain the extension of the history trail in the extent by sorting the blocks in the ascending order of their corresponding access indices. We then prefetch the non-resident ones in the order of their LBNs and place them in the current window, similarly to the sequence-based two-window prefetching. Starting from the last prefetched block, we similarly prefetch blocks into a readahead window. The initial window sizes, or the number of blocks to be prefetched, of these two windows are 8. When the window size is less than min(=8), prefetching aborts. When the window size is larger than max(=64), only the first max blocks are prefetched. If there are multiple matched history trails, we prefetch the intersection of these trails. The two history-aware windows are shifted forward much in the same way as in the sequence-based prefetching. To keep history-aware prefetching enabled, there must be at least one matched history trail. If the history-aware prefetching aborts, sequence-based prefetching is attempted.

3.5 Balancing Memory Allocation between the Prefetching and Caching Areas

In DiskSeen, memory is adaptively allocated between the prefetching area and caching area to maximize system performance, as follows. We extend the reclamation queue with a segment of 2048 blocks which receive the metadata of blocks evicted from the queue. We also set up a FIFO queue, of the same size as the segment for the prefetching area, that receives the metadata of blocks evicted from the caching area. We divide the runtime into epochs, whose size is the period when N_{p-area} disk blocks are requested, where N_{p-area} is a sample of current sizes of the prefetching area in blocks. In each epoch we monitor the numbers of hits to these two segments (actually they are misses in the memory), $H_{prefetch}$ and H_{cache}, respectively. If $|(H_{prefetch} - H_{cache})|/N_{p-area}$ is larger than 10%, we move 128 blocks of memory from the area with fewer hits to the other area to balance the misses between the two.

4 Experimental Evaluation

To evaluate the performance of the DiskSeen scheme in a mainstream operating system, we implemented a prototype in the Linux 2.6.11 kernel. In the following sections we first describe some implementation-related issues, then the experimental results of micro-benchmarks and real-life applications.

4.1 Implementation Issues

Unlike the existing prefetch policies that rely on high-level abstractions (i.e., file ID and offset) that map to disk blocks, the prefetch policy of DiskSeen directly accesses blocks via their disk IDs (i.e., LBNs) without the knowledge of higher-level abstractions. By doing so, in addition to being able to extract disk-specific performance when accessing file contents, the policy can also prefetch metadata, such as inode and directory blocks, that cannot be seen via high-level abstractions, in LBN-ascending order to save disk rotation time. To make the LBN-based prefetched blocks usable by high-level I/O routines, it would be cumbersome to proactively back-translate LBNs to file/offset representations. Instead, we treat a disk partition as a raw device file to read blocks

in a prefetch operation and place them in the prefetching area. When a high-level I/O request is issued, we check the LBNs of requested blocks against those of prefetched blocks. A match causes a prefetched block to move into the caching area to satisfy the I/O request.

To implement the prototype, we added to the stock Linux kernel about 1100 lines of code in 15 existing files concerned with memory management and the file system, and another about 3700 lines in new files to implement the main algorithms of DiskSeen.

4.2 Experimental Setup

The experiments were conducted on a machine with a 3.0GHz Intel Pentium 4 processor, 512MB memory, Western Digital WD1600JB 160GB 7200rpm hard drive. The hard drive has an 8MB cache. The OS is Redhat Linux WS4 with the Linux 2.6.11 kernel using the Ext3 file system. Regarding the parameters for DiskSeen, T, the access index gap threshold, is set as 2048, and S, which is used to determine the trail extent, is set as 128.

4.3 Performance of One-run Benchmarks

We selected six benchmarks to measure their individual run times in varying scenarios. These benchmarks represent various common disk access patterns of interest. Among the six benchmarks, which are briefly described following, *strided* and *reversed* are synthetic and the other four are real-life applications.

1. *strided* is a program that reads a 1GB file in a strided fashion—it reads every other 4KB of data from the beginning to the end of the file. There is a small amount of compute time after each read.

2. *reversed* is a program that sequentially reads one 1GB file from its end to its beginning.

3. *CVS* is a version control utility commonly used in software development environment. We ran *cvs -q diff*, which compares a user's working directory to a central repository, over two identical data sets stored with 50GB space between them.

4. *diff* is a tool that compares two files for character-by-character differences. This was run on two data sets. Its general access pattern is similar to that of CVS. We use their subtle differences to illustrate performance differences DiskSeen can make.

5. *grep* is a tool to search a collection of files for lines containing a match to a given regular expression. It was run to search for a keyword in a large data set.

6. *TPC-H* is a decision support benchmark that processes business-oriented queries against a database system. In our experiment we use PostgreSQL 7.3.18 as the database server. We choose the scale factor 1 to generate the database and run a query against it. We use queries 4 and 17 in the experiment.

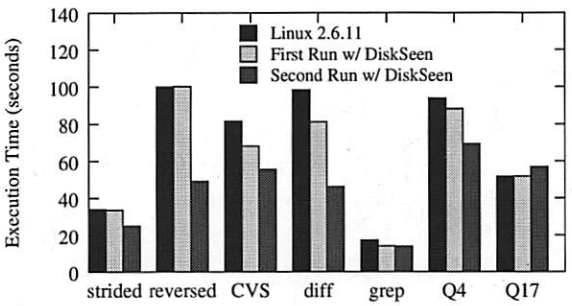


Figure 4: Execution times of the six benchmarks, including two TPC-H queries, Q4 and Q17.

To facilitate the analysis of experiment results across different benchmarks, we use the source code tree of Linux kernel 2.6.11 as the data set, whose size is about 236MB, in benchmarks *CVS*, *diff*, and *grep*. Figure 4 shows the execution times of the benchmarks on the stock Linux kernel, and the times for their first and second runs on the kernel with the DiskSeen enhancement. Between any two consecutive runs, the buffer cache is emptied to ensure all blocks are accessed from disk in the second run. For most of the benchmarks, the first runs with DiskSeen achieve substantial performance improvements due to DiskSeen's sequence-based prefetching, while the second runs enjoy further improvement because of the history information from the first runs. The improved performance for the second runs is meaningful in practice because users often run a program multiple times with only part of the input changed, leaving the on-disk data set accessed as well as access patterns over them largely unchanged across runs. For example, a user may run *grep* many times to search different patterns over the same set of files, or *CVS* or *diff* again with some minor changes to several files. Following we analyze the performance results in detail for each benchmark.

Strided, reversed. With its strided access patterns no sequential access patterns can be detected for *stride* either at the file level or at disk level. The first run with DiskSeen does not reduce its execution time. Neither does it increase its execution time, which shows that the overhead of DiskSeen is minimal. We have a similar observation with *reversed*. With the history information, the second runs of the two benchmarks with DiskSeen show significant execution reductions: 27% for *stride* and 51% for *reversed*, because history trails lead us to find the prefetchable blocks. It is not surprising to see a big improvement with *reversed*. Without prefetching, reversed accesses can cause a full disk rotation time to

service each request. DiskSeen prefetches blocks in large aggregates and requests them in ascending order of their LBNs, and all these blocks can be prefetched in one disk rotation. Note that the disk scheduler has little chance to reverse the continuously arriving requests and service them without waiting for a disk rotation, because it usually works in a work-conserving fashion and requests are always dispatched to disk at the earliest possible time. This is true at least for synchronous requests from the same process. Recognizing that reverse sequential and forward/backward strided accesses are common and performance-critical access patterns in high-performance computing, the GPFS file system from IBM [25] and the MPI-IO standard [19] provide special treatment for identifying and prefetching these blocks. If history access information is available, DiskSeen can handle these access patterns as well as more complicated patterns without making file systems themselves increasingly complex.

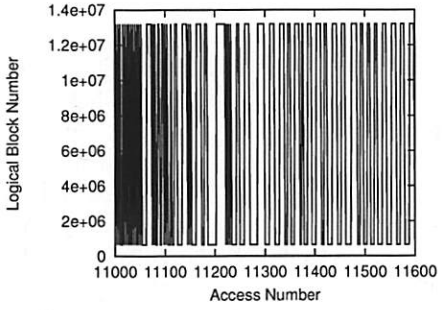


Figure 5: A sample of CVS execution without DiskSeen.

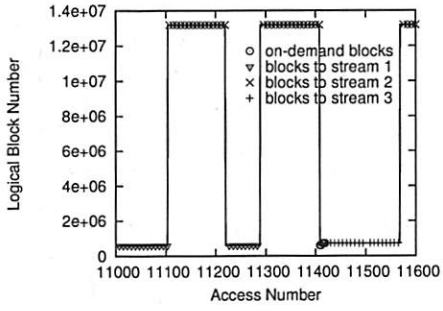


Figure 6: A sample of CVS execution with DiskSeen.

CVS, diff. As shown in Figure 4, DiskSeen significantly improves the performance of both *CVS* and *diff* on the first run and further on the second run. This is because the Linux source code tree mostly consists of small files, and at the file level, sequences across these files cannot be detected, so prefetching is only occasionally activated in the stock Linux kernel. However, many sequences can be detected at the disk level even without history information. Figure 5 shows a segment of *CVS* execution with the stock kernel. The X axis shows the sequence of accesses to disk blocks, and the Y axis shows the LBNs of these blocks. Lines connect points representing consecutive accesses to indicate disk head movements. In comparison, Figure 6 shows the same segment of the second run of *CVS* with DiskSeen. Most of disk head movements between the working directory and the *CVS* repository are eliminated by the disk-level prefetching. The figure also marks accesses of blocks that are on-demand fetched or prefetched into different prefetch streams. It can be seen that there are multiple concurrent prefetch streams, and most accesses are prefetches.

Certainly the radial distance between the directories also plays a role in the *CVS* executions because the disk head must travel for a longer time to read data in the other directory as the distance increases. Figure 7 shows how the execution times of *CVS* with the stock kernel and its runs with DiskSeen would change with the increase in distance. We use disk capacity between the two directories to represent their distance. Although all execution times increase with the increase of the distance, the time for the stock kernel is affected more severely because of the number of head movements involved. For example, when the distance increases from 10GB to 90GB, the time for the original kernel increases by 70%, while the times for first run and second run with DiskSeen increases by only 51% and 36%, respectively.

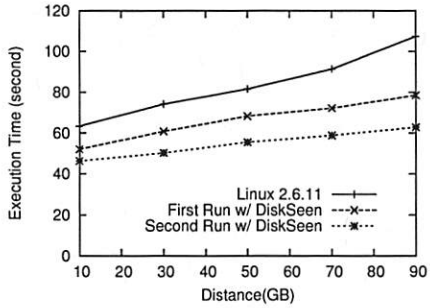


Figure 7: CVS execution times with different directory distances.

While the first runs of *CVS* and *diff* with DiskSeen reduce execution times by 16% and 18%, respectively, the second of them can further reduce the times by another 16% and 36%. For *CVS*, each directory in a *CVS*-managed source tree (i.e., working directory) contains a directory, named as *CVS*, to store versioning information. When *CVS* processes each directory, it first checks the *CVS* subdirectory, then comes back to examine other files/directories in their order in the directory. This visit to the *CVS* subdirectory disrupts the sequential accesses of regular files in the source code tree, and causes a disruption in the sequence-based prefetching. In the second run, new prefetch sequences including the out-of-order blocks (that might not be purely sequential) can be formed by observing history trails. Thus the performance gets further improvement. There are also many non-sequentialities in the execution of *diff* that prevents its first run from exploiting the full performance potential.

When we extract a kernel tar ball, the files/directories in a parent directory are not necessarily laid out in the alphabetical order of their names. However, *diff* accesses these files/directories in strict alphabetical order. So even though these files/directories have been well placed sequentially on disk, these mismatched orders would break many disk sequences, even making accesses in some directories close to random. This is why *diff* has worse performance than *CVS*. Again during the second run, history trails help to find the blocks that are proximate and have been accessed within a relatively short period of time. DiskSeen then sends prefetch requests for these blocks in the ascending order of their LBNs. In this way, the mismatch can be largely corrected and the performance is significantly improved.

Grep: While it is easy to understand the significant performance improvements of *CVS* and *diff* due to their alternate accesses of two remote disk regions, we must examine why *grep*, which only searches a local directory, also has good performance improvement, a 20% reduction in its execution time.

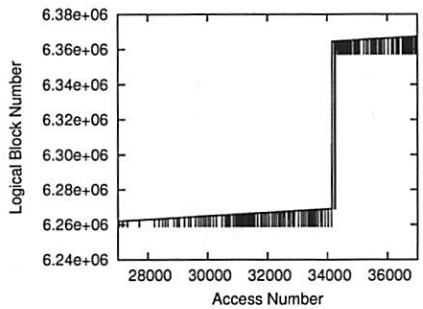


Figure 8: A sample of grep execution without DiskSeen.

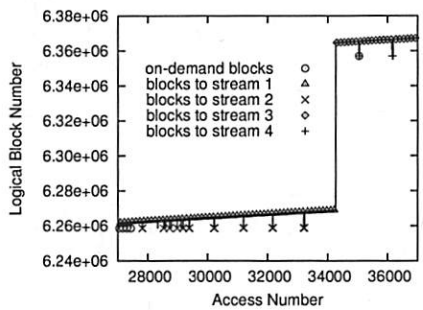


Figure 9: A sample of grep execution with DiskSeen.

Figure 8 shows a segment of execution of *grep* in the stock Linux kernel. This collection of stair-like accesses corresponds to two cylinder groups. In each cylinder group, *inode* blocks are located in the beginning, followed by file data blocks. Before a file is accessed, its *inode* must be inspected, so we see many lines dropping down from file data blocks to *inode* blocks in a cylinder group. Figure 9 shows the corresponding segment of execution of first run of *grep* with DiskSeen. By prefetching *inode* blocks in DiskSeen, most of the disk head movements disappear. The figure also shows that accesses to *inode* blocks and data blocks from different prefetch streams. This is a consequence of the decision to only attempt to prefetch in each localized area.

TPC-H: In this experiment, Query 4 performs a merge-join against table *orders* and table *lineitem*. It sequentially searches table *orders* for records representing orders placed in a specific time frame, and for each such record the query searches for the matched records in table *lineitem* by referring to an index file. Because table *lineitem* was created by adding records generally according to the order time, DiskSeen can identify sequences in each small disk area for prefetching. In addition, history-aware prefetching can exploit history trails for further prefetching opportunities (e.g., reading the index file), and achieve a 26% reduction of execution time compared to the time for the stock kernel.

However, the second run of Q17 with DiskSeen shows performance degradation (a 10% execution increase over the time for the run on the stock kernel). We carefully examined its access pattern in the query and found that table *lineitem* was read in a close-to-random fashion with insignificant spatial locality in many small disk areas. While we used a relatively large access index gap ($T = 2048$) in the experiment, this locality would make history-aware DiskSeen form many prefetch streams, each for a disk area, and prefetch a large number of blocks that will not be used soon. This causes thrashing that even the extended metadata segment of the reclamation queue cannot detect it due to its relative small size. To confirm this observation, we reduced T to 256 and re-ran the query with DiskSeen to which history access information is available. With the reduced T, the execution time is increased by only 2.6%.

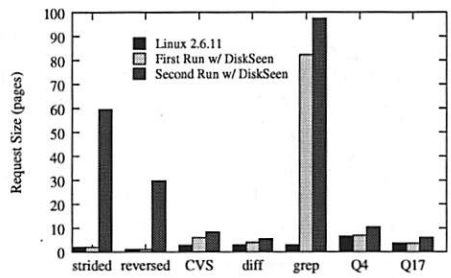


Figure 10: The sizes of requests serviced by disk.

Disk request sizes: Disk performance is directly affected by the sizes of requests a disk receives. To obtain the sizes, we instrument the Linux kernel to monitor READ/WRITE commands issued to the IDE disk controller and record the sizes of corresponding requests. We report the average size of all the requests during the executions of the benchmarks in Figure 10. From the figure we can see that in most cases DiskSeen significantly in-

creases the average request sizes, which corresponds to their respective execution reductions shown in Figure 4. These increases are not proportional to their respective reductions in execution time because of factors such as the proportion of I/O time in the total execution time and differences in the seek times incurred.

4.4 Performance of Continuously Running Application

For applications that are continuously running against the same set of disk data, previous disk accesses could serve as the history access information to improve the I/O performance of current disk accesses. To test this we installed a Web server running the general hypertext cross-referencing tool Linux Cross-Reference (LXR) [15]. This tool is widely used by Linux developers for searching Linux source code.

We use the LXR 0.3 search engine on the Apache 2.0.50 HTTP Server, and use Glimpse 4.17.3 as the free-text search engine. The file set searched is three versions of the Linux kernel source code: 2.4.20, 2.6.11, and 2.6.15. Glimpse divides the files in each kernel into 256 partitions, indexes the file set based on partitions, and generates an index file showing the keyword locations in terms of partitions. The total size of the three kernels and the index files is 896MB. To service a search query, glimpse searches the index file first, then accesses the files included in the partitions matched in the index files. On the client side, we used WebStone 2.5 [29] to generate 25 clients concurrently submitting freetext search queries. Each client randomly picks a keyword from a pool of 50 keywords and sends it to the server. It sends its next query request once it receives the results of its previous query. We randomly select 25 Linux symbols from file */boot/System.map* and another 25 popular OS terms such as *"lru"*, *"scheduling"*, *"page"* as the pool of candidate query keywords. Each keyword is searched in all three kernels. The metric we use is throughput of the query system represented by MBit/sec, which means the number of Mega bits of query results returned by the server per second. This metric is also used for reporting WebStone benchmark results.

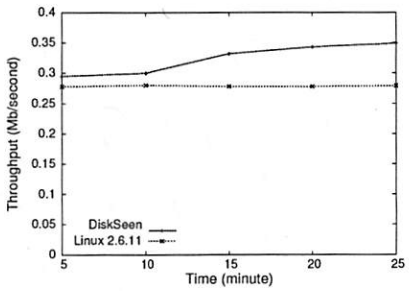


Figure 11: LXR throughputs with and without DiskSeen.

Figure 11 shows the LXR throughputs on the kernels with and without DiskSeen at different times during its execution. We have two observations. First, DiskSeen improves LXR's throughput. This is achieved by prefetching contiguous small files at disk level. Second, from the tenth minute to twentieth minute of the execution, the throughput of LXR with DiskSeen keeps increasing, while the throughput of LXR without DiskSeen does not improve. This demonstrates that DiskSeen can help the application self-improve its performance by using its own accumulated history access information.

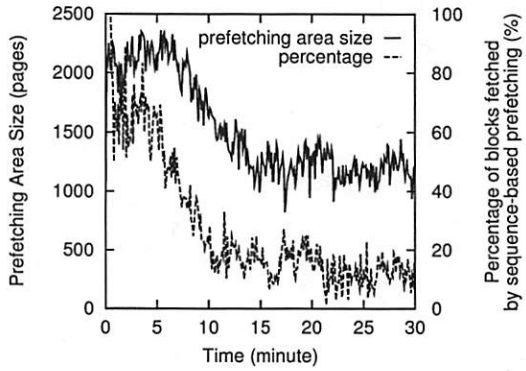


Figure 12: Prefetching area allocation and percentage of blocks fetched by sequence-based prefetching

Figure 12 shows that the size of the prefetching area changes dynamically during execution, and the percentages of blocks that are prefetched through sequence-based prefetching, including the prefetch candidates that are loaded, over all prefetched blocks. We can see that smaller percentages of blocks are loaded through sequence-based prefetching as the application proceeds, i.e., a larger percentage of blocks are loaded through history-aware prefetching, because of the availability of history information. This trend corresponds to the reduction of the prefetching area size. History-aware prefetching has higher accuracy than sequence-based prefetching (the miss ratios of history-aware prefetching and sequence-based prefetching are 5.2% and 11%, respectively), and most blocks fetched by history prefetching are hits and are moved to the caching area shortly after they enter the prefetching area. Thus, there are fewer hits to the metadata segment extended from the reclamation queue in the prefetching area. Accordingly, DiskSeen adaptively re-allocates some buffer space used by the prefetching area to the caching area.

4.5 Interference of Noisy History

While well matched history access information left by prior run of applications is expected to provide accurate hints and improve performance, a reasonable speculation is that a misleading history could confuse DiskSeen and even direct DiskSeen to prefetch wrong blocks so as to

cause DiskSeen to actually degrade application performance. To investigate the interference effect caused by *noisy* history on DiskSeen's performance, we designed experiments in which two applications access the same set of data with different access patterns. We use *grep* and *diff* as test applications. *Grep* searches a keyword in a Linux source code tree, which is also used by *diff* to compare against another Linux source code tree. We know that *grep* scans files basically in the order of their disk layout, but *diff* visits files in the alphabetic order of directory/file names.

In the first two experiments, we run the applications alternatively, specifically in sequence (*diff*, *grep*, *diff*, *grep*) in experiment I and sequence (*grep*, *diff*, *grep*, *diff*) in experiment II. Between any two consecutive runs, the buffer cache is emptied to ensure the second run does not benefit from cached data while history access information in the block table is passed across a sequence of runs in an experiment. The execution times compared to the stock kernel are shown in Table 1.

Experiment	Execution times (seconds)			
Linux	*diff*	98.4	*grep*	17.2
I	*diff*	*grep*	*diff*	*grep*
	81.1	16.3	46.4	14.0
II	*grep*	*diff*	*grep*	*diff*
	14.0	67.7	13.9	46.1
Linux	*grep/diff*		20.9/55.8	
III	*grep/diff*		15.2/44.9	
			17.0/34.6	
			17.9/34.8	
			18.2/34.5	
			18.3/35.1	

Table 1: Execution times for *diff* and *grep* when they are alternately executed in different orders or concurrently, with DiskSeen, compared to the times for the stock kernel. The times reported are wall clock times.

If we use the execution times without any history as reference points (the first runs in experiments I and II), where only sequence-based prefetching occurs, noisy history causes the degradation of performance in the first run of *grep* by 16% (14.0s vs. 16.3s) in experiment I, while it accidentally helps improve the performance in the first run of *diff* by 17% (81.1s vs. 67.7s) in experiment II. The degradation in experiment I is due to the history access information left by *diff* that misleads DiskSeen, which is running *grep*, to infer that a matched history trail has been found and initiate a history-based prefetching. However, the matched history trail is broken when *diff* takes a different order to visit files. This causes DiskSeen to fall back to its sequence-based prefetching, which takes some time to be activated (accesses of 8 contiguous blocks). Thus, history-aware prefetching attempts triggered by noisy history keep sequence-based prefetching from achieving its performance potential. It is interesting to see that a trail left by *grep* improves the performance of *diff*, which has a different access pattern, in Experiment II. This is because the trails left by *grep* are also sequences on disk. Using these trails for history-aware prefetching essentially does not change the behavior of sequence-based prefetching, except that the prefetching becomes more aggressive, which helps reduce *diff*'s execution time. For the second runs of *grep* or *diff* in either experiment, the execution times are very close to those of the second runs shown in Figure 4. This demonstrates that noisy history only very slightly interferes with history-aware prefetching if there also exists a well-matched history in the block table (e.g., the ones left by the first runs of *grep* or *diff*, respectively).

In the third experiment, we concurrently ran these two applications five times, with the times of each run reported in Table 1, along with their counterparts for the stock kernel. The data shared by *diff* and *grep* are fetched from disk by whichever application first issues requests for them, and requests for the same blocks from the other application are satisfied in memory. The history of the accesses of the shared blocks is the result of mixed requests from both applications. Because of the uncertainty in process scheduling, access sequences cannot be exactly repeated between different runs. Each run of the two applications leaves different access trails on the shared blocks, which are noisy history that interferes with the current DiskSeen prefetching. The more runs there have been, the more history is recorded, the easier it is to trigger an incorrect history-aware prefetching. This is why the execution time of *grep* keeps increasing until the fifth run (we keep at most four access indices for each block). Unlike *grep*, the execution time of *diff* in the second run is decreased by 23% (34.6s over 44.9s). This is because history-aware prefetching of the other source code tree, which is not touched by *grep*, is not affected by the interference.

4.6 DiskSeen with a Contrived Adverse Workload

To demonstrate the extent to which DiskSeen could be ill-behaved, we designed an arguably worst-case scenario in which all predictions made by history-aware prefetching are wrong. In the experiment, a 4GB file was divided into chunks of 20 4KB blocks. Initially we sequentially read the file from its beginning to create a corresponding sequential trail. After removing buffered blocks of the file from memory, we read four blocks at the beginning of each chunk, chunk by chunk from the beginning to the end of the file. The access of four blocks in a chunk triggers a history-aware prefetching, which prefetches two

windows, each of 8 blocks, in the same chunk. These 16 blocks in each chunk are all mis-predicted. The experimental result shows that for the second file read DiskSeen increased the execution time by 3.4% (from 68.0 seconds in the stock kernel to 70.3 seconds with DiskSeen). The small increase is due to the sequential access of chunks, in which the disk head will move over the prefetched blocks whether or not prefetch requests are issued. To eliminate this favorable scenario, we randomly accessed the chunks in the second read, still with only four blocks requested from each chunk. This time DiskSeen increased the execution time by 19% (from 317 seconds in the stock kernel to 378 seconds with DiskSeen), which represents a substantial performance loss. However, this scenario of a slowdown of more than fourfold (for either scheme) could often be avoided at the application level by optimizing large-scope random accesses into sequential accesses or small-region random accesses.

4.7 Discussion and Future Work

From the benchmarking we have conducted, DiskSeen is most effective in transforming random or semi-random accesses that take place on one or more limited disk areas into (semi-)sequential accesses in each disk locality. It is also effective in discovery and exploitation of sequential on-disk access that is difficult to detect at the file level.

We have not implemented a prefetch throttling mechanism in DiskSeen. This makes our system incapable of responding to overly-aggressive prefetching that leads to thrashing (e.g., in the case of Q17 of TPC-H) and miss-prefetching (e.g., in the case described in Section 4.6). An apparent fix to the issue would be a policy that adaptively adjusts the access index gap (T) based on the effectiveness of recent prefetchings (i.e., the percentage of blocks prefetched by the history-aware approach that were subsequently used). However, in a system where applications of various access patterns run concurrently, the adjustment may have to be made differently for different applications, or different access index gaps need to be used. While our experiments suggest that a fixed T works well for most access patterns and its negative impact is limited, we leave a comprehensive investigation of the issue as our future work.

There are several limitations in our work to be addressed in the future. First, our implementation and performance evaluations are currently based on one disk drive. Most enterprise-level storage systems are composed of RAIDs and their associated controllers. While we expect that DiskSeen can retain most of its performance merits because the mappings between logical blocks and the physical blocks on multiple disks still maintain high performance for contiguous LBN accesses, some new issues have to be addressed, such as the conditions on which prefetching should cross the disk boundary and the relationship between prefetching aggressiveness and parallelism of RAID. Second, we have evaluated the prototype only in a controlled experimental setting. It would be worthwhile to evaluate the system in a real-world environment with mixed workloads running for extended periods, such as using it on a file server that supports programming projects of a class of students or an E-business service. Third, the block table could become excessively large. For example, streaming of data from an entire 500GB disk drive can cause the table grow to 2GB. In this case, we need to page out the table to the disk. Other solutions would be compression of the table or avoidance of recording streaming access in the table.

5 Related Work

There are several areas of effort related to this work, spanning applications, operating systems, and file systems.

Intelligent prefetching algorithms: Prefetching is an actively research area for improving I/O performance. Operating systems usually employ sophisticated heuristics to detect sequential block accesses to activate prefetching, as well as adaptively adjust the number of blocks to be prefetched within the scope of a single file [20, 22]. By working at the file abstraction and lacking mechanism for recording historically detected sequential access patterns, the prefetch policies usually make conservative predictions, and so may miss many prefetching opportunities [21]. Moreover, their predictions cannot span files.

There do exist approaches that allow prefetching across files. In these approaches, system-wide file access history has been used in probability-based prediction algorithms, which track sequences of file access events and evaluate the probability of file occurrences in the sequences [9, 13]. These approaches may achieve a high prediction accuracy via their use of historical information. However, the prediction and prefetching are built on the unit of files rather than file blocks, which makes the approaches more suitable to web proxy/server file prefetching than to the prefetching in general-purpose operating systems [6]. The complexity and space costs have also thus far prevented them from being deployed in general-purpose operating systems. Moreover, these approaches are not applicable to prefetching for disk paging in virtual memory and file metadata.

Hints from applications: Prefetching can be made more effective with hints given by applications. In the TIP project, applications disclose their knowledge of future I/O accesses to enable informed caching and prefetching [18, 27]. The requirements on hints are usually high—they are expected to be detailed and to be given early enough to be useful. There are some other

buffer cache management schemes using hints from applications [3, 5].

Compared with the method used in DiskSeen, application-hinted prefetching has limitations: (1) The requirements for generating detailed hints may put too much burden on application programmers, and could be infeasible. As an example, a file system usage study for Windows NT shows that only 5% of file-opens with sequential reads actually take advantage of the option for indicating their sequential access pattern to improve I/O performance [28]. Another study conducted at Microsoft Research shows a consistent result [7]. It would be a big challenge to require programmers to provide detailed hints sometimes by even restructuring the programs, as described in the papers on TIP [18, 27]. The DiskSeen scheme, in contrast, is transparent to applications. (2) The sequentiality across files and the sequentiality of data disk locations still cannot be disclosed by applications, which are important for prefetching of small files. In our work this sequentiality can be easily detected and exploited.

Prefetching hints can also be automatically abstracted by compilers [16] or generated by OS-supported speculative executions [4, 8]. Another interesting work is a tool called *C-Miner* [14], which uses a data mining technique to infer block correlations by monitoring disk block access sequences. The discovered correlations can be used to determine prefetchable blocks. Though the performance benefits of these approaches can be significant, they do not cover the benefits gained from simultaneously exploiting temporal and spatial correlations among on-disk blocks. In a sense, our work is complementary.

Improving data placement: Exposing information from the lower layers up for better utilization of hard disk is an active research topic. Most of the work focuses on using disk-specific knowledge for improving data placements on disk that facilitate the efficient servicing of future requests. For example, Fast File System (FFS) and its variants allocate related data and metadata into the same cylinder group to minimize seeks [17, 10]. Traxtent-aware file system excludes track boundary block from being allocated for better disk sequential access performance [24]. However, these optimized block placements cannot be seen at the file abstraction. Because most files are of small sizes (e.g., a study on Windows NT file system usage shows that 40% of operations are to files shorter than 2KB [28]), prefetching based on individual file abstractions cannot take full advantages of these efforts. In contrast, DiskSeen can directly benefit from these techniques by being able to more easily find sequences that can be efficiently accessed based on optimized disk layout.

Recently, the FS2 file system was proposed to dynamically create block replicas in free spaces on disk according to the observed disk access patterns [11]. These replicas can be used to provide faster accesses of disk data. FS2 dynamically adjusts disk data layout to make it friendly to the changing data request pattern, while DiskSeen leverages buffer cache management to create disk data request patterns that exploit current disk layout for high bandwidth. These two approaches are complementary. Compared with looking for free disk space to make replicas consistent to the access patterns in FS2, DiskSeen can be more flexible and responsive to the changing access pattern.

6 Conclusions

DiskSeen addresses a pressing issue in prefetch techniques—how to exploit disk-specific information so that effective disk performance is improved. By efficiently tracking disk accesses both in the live request stream and recorded prior requests, DiskSeen performs more accurate block prefetching and achieves more continuous streaming of data from disk by following the block number layout on the disk. DiskSeen overcomes barriers imposed by file-level prefetching such as the difficulties in relating accesses across file boundaries or across lifetimes of open files. At the same time, DiskSeen complements rather than supplants high-level prefetching schemes. Our implementation of the DiskSeen scheme in the Linux 2.6 kernel shows that it can significantly improve the effectiveness of prefetching, reducing execution times by 20%-53% for micro-benchmarks and real applications such as *grep*, *CVS*, *TPC-H*, and LXR.

7 Acknowledgements

We are grateful to Dr. Fay Chang for her detailed comments and suggestions on the final version of the paper. We thank the anonymous reviewers for their constructive comments. This research was supported in part by National Science Foundation grants CNS-0405909 and CCF-0602152.

References

[1] Journaling-Filesystem Fragmentation Project, *URL: http://www.informatik.uni-frankfurt.de/ loizides/reiserfs/agesystem.html*

[2] A. R. Butt, C. Gniady, and Y. C. Hu, "The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms", *in Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*, June 2005.

[3] P. Cao, E. W. Felten, A. Karlin and K. Li, "Implementation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling",

ACM Transaction on Computer Systems, November 1996.

[4] F.W. Chang and G.A. Gibson, "Automatic I/O Hint Generation through Speculative Execution", *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, February 1999.

[5] P. Cao, E. W. Felten and K. Li, "Application-Controlled File Caching Policies", *Proceedings of the USENIX Summer 1994 Technical Conference*, 1994.

[6] X. Chen and X. Zhang, "A popularity-based prediction model for Web prefetching", *IEEE Computer*, Vol. 36, No. 3, March 2003.

[7] J. R. Douceur and W. J. Bolosky, "A Large-Scale Study of File-System Contents", *Proceedings of the 1999 ACM SIGMETRICS conference*, May 1999.

[8] K. Fraser and F. Chang, "Operating system I/O Speculation: How two invocations are faster than one", *Proceedings of the USENIX Annual Technical Conference* June 2003.

[9] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach", *Proceedings of the Usenix Summer Conference*, June 1994, pp. 197-208.

[10] G. Ganger and F. Kaashoek, "Embedded Inodes and Explicit Groups: Exploiting Disk Bandwidth for Small Files", *Proceedings of the 1997 USENIX Annual Technical Conference*, January 1997.

[11] H. Huang, W. Hung, and K. G. Shin, "FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption", *Proceedings of 20th ACM Symposium on Operating Systems Principles*, October 2005.

[12] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: an Effective Buffer Cache Management Scheme to Exploit both Temporal and Spatial Locality", *Proceedings of the 4th USENIX Conference on File and Storage Technology (FAST'05)*, December 2005.

[13] T. M. Kroeger and D.D.E. Long, "Design and implementation of a predictive file prefetching algorithm", *Proceedings of the 2001 USENIX Annual Technical Conference*, January 2001.

[14] Z. Li, Z. Chen, S. Srinivasan and Y. Zhou, "C-Miner: Mining Block Correlations in Storage Systems", *Proceedings of 3rd USENIX Conference on File and Storage Technologies (FAST04)*, March 2004.

[15] Linux Cross-Reference, *URL : http://lxr.linux.no/.*

[16] T. C. Mowry, A. K. Demke and O. Krieger. "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications", *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*,, October 1996.

[17] M. K. Mckusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX", *Transactions on Computer Systems*, 2(3), 1984.

[18] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky and J. Zelenka, "Informed Prefetching and Caching", *Proceedings of the 15th Symposium on Operating System Principles*, 1995, pp. 1-16.

[19] MPI-2: Extensions to the Message-Passing Interface, *URL : http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html*

[20] R. Pai, B. Pulavarty, and M. Cao, "Linux 2.6 Performance Improvement through Readahead Optimization", *Proceedings of the Linux Symposium*, July 2004.

[21] A. E. Papathanasiou and M. L. Scott, "Aggressive Prefetching: An Idea Whose Time Has Come", *Proceedings of the Tenth Workshop on Hot Topics in Operating Systems*, June 2005.

[22] A. J. Smith, "Sequentiality and Prefetching in Database Systems", *ACM Trans. on Database Systems*, Vol. 3, No. 3, 1978, pp. 223-247.

[23] J. Schindler and G. R. Ganger, "Automated Disk Drive Characterization", *Proceeding of 2000 ACM SIGMETRICS Conference*, June 2000.

[24] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger, "Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics", *USENIX Conference on File and Storage Technologies (FAST)*, January 2002.

[25] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters", *USENIX Conference on File and Storage Technologies (FAST)*, January 2002.

[26] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger, "On Multidimensional Data and Modern Disks", *Proceedings of the 4th USENIX Conference on File and Storage Technology (FAST'05)*, December 2005.

[27] A. Tomkins, R. H. Patterson and G. Gibson, "Informed Multi-Process Prefetching and Caching", *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997.

[28] W. Vogels, "File system usage in Windows NT 4.0", *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[29] WebStone — The Benchmark for Web Servers, *URL : http://www.mindcraft.com/benchmarks/webstone/*

[30] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes, "On-line extraction of SCSI disk drive parameters" *In Proceeding of 1995 ACM SIGMETRICS Conference*, May 1995.

Notes

[1] We make this statement for generic OS kernels. Some operating systems adopt aggressive prefetch policies which rely on high-level knowledge about user/application behaviors. An example is the SuperFetch technique in Windows Vista, which performs prefetching according to particular applications, users, usage times of day or even usage days of week.

[2] Specifically we do not expose information about logical disk layout, which actually has been available for prefetch operations in operating systems. We use 'expose' to indicate a general approach utilizing low-level disk-specific knowledge, which could include hidden disk geometry information below the LBN abstraction in future work.

[3] In the implementation the prefetch streams are only a conceptual data structure—they are embedded in the reclamation queue and blocks appear only once.

A Memory Soft Error Measurement on Production Systems*

Xin Li Kai Shen Michael C. Huang Lingkun Chu
University of Rochester *Ask.com*
{xinli@ece, kshen@cs, huang@ece}.rochester.edu lchu@ask.com

Abstract

Memory state can be corrupted by the impact of particles causing single-event upsets (SEUs). Understanding and dealing with these soft (or transient) errors is important for system reliability. Several earlier studies have provided field test measurement results on memory soft error rate, but no results were available for recent production computer systems. We believe the measurement results on real production systems are uniquely valuable due to various environmental effects. This paper presents methodologies for memory soft error measurement on production systems where performance impact on existing running applications must be negligible and the system administrative control might or might not be available.

We conducted measurements in three distinct system environments: a rack-mounted server farm for a popular Internet service (Ask.com search engine), a set of office desktop computers (Univ. of Rochester), and a geographically distributed network testbed (PlanetLab). Our preliminary measurement on over 300 machines for varying multi-month periods finds 2 suspected soft errors. In particular, our result on the Internet servers indicates that, with high probability, the soft error rate is at least two orders of magnitude lower than those reported previously. We provide discussions that attribute the low error rate to several factors in today's production system environments. As a contrast, our measurement unintentionally discovers permanent (or hard) memory faults on 9 out of 212 Ask.com machines, suggesting the relative commonness of hard memory faults.

1 Introduction

Environmental noises can affect the operation of microelectronics to create soft errors. As opposed to a "hard" error, a soft error does not leave lasting effects once it is corrected or the machine restarts. A primary noise mechanism in today's machines is particle strike. Particles hitting the silicon chip create electron-hole pairs which, through diffusion, can collect at circuit nodes and outweigh the charge stored and create a flip of logical state, resulting in an error. The soft error problem at sea-level was first discovered by Intel in 1978 [9].

Understanding the memory soft error rate is an important part in assessing whole-system reliability. In the presence of inexplicable system failures, software developers and system administrators sometimes point to possible occurrences of soft errors without solid evidence. As another motivating example, recent studies have investigated the influence of soft errors on software systems [10] and parallel applications [5], based on presumably known soft error rate and occurrence patterns. Understanding realistic error occurrences would help quantify the results of such studies.

A number of soft error measurement studies have been performed in the past. Probably the most extensive test results published were from IBM [12, 14–16]. Particularly in a 1992 test, IBM reported 5950 FIT (Failures In Time, specifically, errors in 10^9 hours) of error rate for a vendor 4Mbit DRAM. The most recently published results that we are aware of were based on tests in 2001 at Sony and Osaka University [8]. They tested 0.18 μm and 0.25 μm SRAM devices to study the influence of altitude, technology, and different sources of particles on the soft error rate, though the paper does not report any absolute error rate. To the best of our knowledge, Normand's 1996 paper [11] reported the only field test on production systems. In one 4-month test, they found 4 errors out of 4 machines with total 8.8 Gbit memory. In another 30-week test, they found 2 errors out of 1 machine with 1 Gbit memory. Recently, Tezzaron [13] collected error rates reported by various sources and concluded that 1000–5000 FIT per Mbit would be a reasonable error rate for modern memory devices. In summary, these studies all suggest soft error rates in the range of 200–5000 FIT per Mbit.

Most of the earlier measurements (except [8]) were over a decade old and most of them (except [11]) were conducted in artificial computing environments where the target devices are dedicated for the measurement. Given the scaling of technology and the countermeasures deployed at different levels of system design, the trends of error rate in real-world systems are not clear. Less obvious environmental factors may also play a role. For example, the way a machine is assembled and packaged as well as the memory chip layout on the main computer board can affect the

*This work was supported in part by the National Science Foundation (NSF) grants CCR-0306473, ITR/IIS-0312925, CNS-0509270, CNS-0615045, and CCF-0621472. Shen was also supported by an NSF CAREER Award CCF-0448413 and an IBM Faculty Award.

chance of particle strikes and consequently the error rate.

We believe it is desirable to measure memory soft errors in today's representative production system environments. Measurement on production systems poses significant challenges. The infrequent nature of soft errors demands long-term monitoring. As such, our measurement must not introduce any noticeable performance impact on the existing running applications. Additionally, to achieve wide deployment of such measurements, we need to consider the cases where we do not have administrative control on measured machines. In such cases, we cannot perform any task requiring the privileged accesses and our measurement tool can be run only at user level. The rest of this paper describes our measurement methodology, deployed measurements in production systems, our preliminary results and the result analysis.

2 Measurement Methodology and Implementation

We present two soft error measurement approaches targeting different production system environments. The first approach, memory controller direct checking, requires administrative control on the machine and works only with ECC memory. The second approach, non-intrusive user-level monitoring, does not require administrative control and works best with non-ECC memory. For each approach, we describe its methodology, implementation, and analyze its performance impact on existing running applications in the system.

2.1 Memory Controller Direct Checking

An ECC memory module contains extra circuitry storing redundant information. Typically it implements single error correction and double error detection (SEC-DED). When an error is encountered, the memory controller hub (a.k.a. Northbridge) records necessary error information in some special-purpose registers. Meanwhile, if the error involves a single bit, then it is corrected automatically by the controller. The memory controller typically signals the BIOS firmware when an error is discovered. The BIOS error-recording policies vary significantly from machine to machine. In most cases, single-bit errors are ignored and never recorded. The BIOS typically clears the error information in memory controller registers on receiving error signals. Due to the BIOS error handling, the operating system would not be directly informed of memory errors without reconfiguring the memory controller.

Our memory controller direct checking of soft errors includes two components:

- *Hardware configuration*: First, we disable any BIOS manipulation of the memory controller error handling in favor of error handling by the OS software. Particularly, we do not allow BIOS to clear memory controller error information. Second, we enable periodic hardware-level memory scrubbing which walks through the memory space to check errors. This is in addition to error detection triggered by software-initiated memory reading. Errors discovered at the hardware level are recorded in appropriate memory controller registers. Memory scrubbing is typically performed at a low frequency (e.g., 1 GB per 1.5 hours) to minimize its energy consumption and interruption to running applications.
- *Software probing*: We augment the OS to periodically probe appropriate memory controller registers and acquire desired error information. Since the memory controller register space is limited and usually only a few errors are recorded, error statistics can be lost if the registers are not read and cleared in time. Fortunately, soft error is typically a rare event and thus our probing can be quite infrequent — it only needs to be significantly more often than the soft error occurrence frequency.

Both hardware configuration and software probing in this approach require administrative privilege. The implementation involves modifications to the memory controller driver inside the OS kernel. The functionality of our implementation is similar to the Bluesmoke tool [3] for Linux. The main difference concerns exposing additional error information for our monitoring purpose.

In this approach, the potential performance impact on existing running applications includes the software overhead of controller register probing and memory bandwidth consumption due to scrubbing. With low frequency memory scrubbing and software probing, this measurement approach has a negligible impact on running applications.

2.2 Non-intrusive User-level Monitoring

Our second approach employs a user-level tool that transparently recruits memory on the target machine and periodically checks for any unexpected bit flips. Since our monitoring program competes for the memory with running applications, the primary issue in this approach is to determine an appropriate amount of memory for monitoring. Recruiting more memory makes the monitoring more effective. However, we must leave enough memory so that the performance impact on other running applications is limited. This is important since we target production systems hosting real live applications and our monitoring must be long running to be effective.

This approach does not require administrative control on the target machine. At the same time, it works best with non-ECC memory since the common SEC-DED feature in ECC memory would automatically correct single-bit errors

and consequently our user-level tool cannot observe them.

Earlier studies like Acharya and Setia [1] and Cipar et al. [4] have proposed techniques to transparently steal idle memory from non-dedicated computing facilities. Unlike many of the earlier studies, we do not have administrative control of the target system and our tool must function completely at user level. The system statistics that we can use are limited to those explicitly exposed by the OS (e.g., the Linux `/proc` file system) and those that can be measured by user-level micro-benchmarks [2].

Design and Implementation Our memory recruitment for error monitoring should not affect the performance of other running applications. We can safely recruit the free memory pages that are not allocated to currently running applications. Furthermore, some already allocated memory may also be recruited as long as we can bound its effects on existing running applications. Specifically, we employ a *stale-memory recruitment policy* in which we only recruit allocated memory pages that have not been recently used (i.e., not used for a certain duration of time D). Under this policy, within any time period of duration D, every physical memory page can be recruited for no more than once (otherwise the second recruitment would have recruited a page that was recently allocated and used). Therefore, within any time period of duration D, the additional application page evictions induced by our monitoring is bounded by the physical memory size S. If we know the page fault I/O throughput R, we can then bound the application slowdown induced by our monitoring to $\frac{S}{D \cdot R}$.

Below we present a detailed design to our approach, which includes three components.

- *Memory recruitment*: We periodically invoke a routine to recruit memory to the monitoring pool. First, it checks the amount of free memory in the system that is not used by any running applications (through existing system interface or a user-level micro-benchmark). In order to further utilize those allocated but rarely used memory, we may also recruit a certain amount of extra memory in the absence of memory contention. Memory contention can be detected by observing recent eviction of pages from the monitoring pool (see below) or a slowdown of memory-intensive tasks (motivated by [6]).
- *Periodic touching*: Since our monitoring tool runs as a normal user-level program, it competes memory with other running applications according to the OS memory management. We assume that the OS employs the Least-Recently Used (LRU) page replacement order or its approximation. To realize the stale-memory recruitment policy, we periodically access the recruited memory pages so that each page is reused at the frequency of once per time duration D. Under the LRU replacement order, application pages that are accessed more often are unlikely to be evicted before the recruited pages in our monitoring tool.

  The touching also serves the purpose of error checking. We read every single word of the page and examines if the pattern written initially still remains. If not, it indicates an error just occurred in the most recent period.
- *Releasing evicted pages*: Recruited memory pages may be swapped out of physical memory during memory contention when all existing application pages are not stale enough (i.e., have been used within the last time duration D). We should detect these evicted pages and release them from the monitoring pool.

We discuss some implementation issues in practice. First, the OS typically attempts to maintain a certain minimum amount of free memory (e.g., to avoid deadlocks when reclaiming pages) and a reclamation is triggered when the free memory amount falls below the threshold (we call `minfree`). We can measure `minfree` of a particular system by running a simple user-level micro-benchmark. At the memory recruitment, we are aware that the practical free memory in the system is the nominal free amount subtract `minfree`.

Second, it may not be straightforward to detect evicted pages from the monitoring pool. Some systems provide direct interface to check the in-core status of memory pages (e.g., the `mincore` system call). Without such direct interface, we can tell the in-core status of a memory page by simply measuring the time of accessing any data on the page. Note that due to OS prefetching, the access to a single page might result in the swap-in of multiple contiguous out-of-core pages. To address this, each time we detect an out-of-core recruited page, we discard several adjacent pages (up to the OS prefetching limit) along with it.

Performance Impact on Running Applications We measure the performance impact of our user-level monitoring tool on existing running applications. Our tests are done in a machine with a 2.8 GHz Pentium4 processor and 1 GB main memory. The machine runs Linux 2.6.18 kernel. We examine three applications in our test: 1) the Apache web server running the static request portion of the SPECweb99 benchmark; 2) MCF from SPEC CPU2000 — a memory-intensive vehicle scheduling program for mass transportation; and 3) compilation and linking of the Linux 2.6.18 kernel. The first is a typical server workload while the other two are representative workstation workloads.

We set the periodic memory touching interval D according to a desired application slowdown bound. An accurate setting requires the knowledge of the page fault

I/O throughput. Here we use a simple estimation of I/O throughput as half the peak sequential disk access throughput (around 57 MB/s for our disk). Therefore a periodic memory touching interval D=0.48 minutes is needed for achieving 2% application slowdown bound. Our program was able to recruit 376.70 MB, 619.24 MB and 722.77 MB on average (out of the total 1 GB) when it runs with Apache, MCF, and Linux compilation respectively. At the same time, the slowdown is 0.52%, 0.26%, and 1.20% for the three applications respectively. The slowdown for Apache is calculated as "$1 - \frac{\text{new throughput}}{\text{original throughput}}$" while the slowdown for the other two applications is calculated as "$1 - \frac{\text{original runtime}}{\text{new runtime}}$". The monitoring-induced slowdown can be reduced by increasing the periodic memory touching interval D. Such adjustment may at the same time reduce the amount of recruited memory.

2.3 Error Discovery in Accelerated Tests

To validate the effectiveness of our measurement approaches and resulted implementation, we carried out a set of accelerated tests with guaranteed error occurrences. To generate soft errors, we heated the memory chip using a heat gun. The machine under test contains 1 GB DDR2 memory with ECC and the memory controller is Intel E7525 with ECC. The ECC feature has a large effect on our two approaches — the controller direct checking requires ECC memory while the user-level monitoring works best with non-ECC memory. To consider both scenarios, we provide results for two tests — one with the ECC hardware enabled and the other with ECC disabled. The results on error discovery are shown in Table 1.

Overall, results suggest that both controller direct probing and user-level monitoring can discover soft errors at respective targeted environments. With ECC enabled, all the single-bit errors are automatically corrected by the ECC hardware and thus the user-level monitoring cannot observe them. We also noticed that when ECC was enabled, the user-level monitoring found less multi-bit errors than the controller direct checking did. This is because the user-level approach was only able to monitor part of the physical memory space (approximately 830 MB out of 1 GB).

Test with ECC enabled

Approach	Single-bit errors	Multi-bit errors
Controller checking	2472	139
User-level monitoring	N/A	106

Test with ECC disabled

Approach	Single-bit errors	Multi-bit errors
User-level monitoring	15	0

Table 1: Errors found in heat-induced accelerated tests. The ECC-disabled test was done at much lower heat intensity compared to the ECC-enabled one. We did this because without the shielding from ECC, all single-bit errors may manifest at the software level, and thus high heat intensity is very likely to crash the OS and disrupt the test.

3 Deployed Measurements and Preliminary Results

We have deployed our measurement in three distinct production system environments: a rack-mounted server farm, a set of office desktop computers, and a geographically distributed network testbed. We believe these measurement targets represent many of today's production computer system environments.

- *Ask.com servers:* These rack-mounted servers are equipped with high-end ECC memory modules and we have administrative control over these machines. We use the memory controller direct checking approach in this measurement. We monitored 212 servers for approximately three months. On average, we were monitoring 3.92 GB memory on each machine.
- *UR desktop computers:* We conducted measurement on a number of desktop computers at the University of Rochester. These machines are provided on the condition of no change to the system and no impact to the running application performance. Therefore we employ the user-level monitoring approach in this measurement. Since this approach works best with non-ECC memory, we identified a set of machines with non-ECC memory by checking vendor-provided machine specification. This measurement has been deployed on 20 desktop (each with 512 MB RAM) computers for around 7 months. On average we recruited 104.23 MB from each machine.
- *PlanetLab machines:* We chose PlanetLab because of its geographically distributed set of machines. Geographic locations (and elevation in particular) may affect soft error occurrence rate. Since we do not have administrative control for these machines, we employ the user-level monitoring approach in this measurement. Again, we search for a set of machines with non-ECC memory to maximize the effect of our measurement. Since we do not know the exact models of most PlanetLab machines, we cannot directly check the vendor-provided machine specification. Our approach is to collect memory controller device ID from the `/proc` file system and then look up its ECC capability. This measurement has been deployed on 70 PlanetLab machines for around 7 months. Since most PlanetLab machines are intensively used and free memory is scarce, we were only able to recruit

Measurement environment	Ask.com	Ask.com (excluding 9 servers with hard errors)	UR Desktops	PlanetLab
Time-memory extent	76,456 GB×day	73,571 GB×day	428 GB×day	23 GB×day
Measurement result	8288 errors, most of which are believed to be hard errors	2 errors, suspected to be soft errors	no error	no error

Table 2: Results of deployed measurements.

approximately 1.54 MB from each machine.

Aside from the respective pre-deployment test periods, we received no complaint on application slowdown for all three measurement environments.

So far we detected no errors on UR desktop computers and PlanetLab machines. At the same time, our measurements on Ask.com servers logged 8288 memory errors concentrating on 11 (out of 212) servers. These errors on the Ask.com servers warrant more explanations:

- Not all logged errors are soft errors. In particular, some are due to permanent (hard) chip faults. One way to distinguish soft and hard errors is that hard errors tend to repeat on the same memory addresses since they are not correctable. On the other hand, soft errors rarely repeat on the same memory addresses with the assumption that soft errors occur on memory addresses in a uniformly random fashion. Using this criterion, we find 9 out of these 11 severs contain hard chip faults.
- We assume that soft errors and hard errors occur independently on host machines. We believe the assumption is reasonable because soft errors are typically due to external environmental factors (e.g., particle strikes) while hard errors are largely due to internal chip properties. With this assumption, we can exclude the 9 machines with known hard errors from our Ask.com measurement pool without affecting the representativeness of soft error statistics on the remaining machines.
- After excluding the 9 servers with known hard errors, there are 2 machines each with a suspected memory soft error.
- Most detected errors on the 11 Ask.com servers are single-bit errors correctable by the ECC, and thus pose no impact on running software. However, the logged memory controller information suggests that at least one machine contains some multi-bit errors that are not correctable.

In Table 2, we list the overall *time-memory extent* (defined as the product of time and the average amount of considered memory over time) and discovered errors for all deployed measurement environments.

Result Analysis Based on the measurement results, below we calculate a probabilistic upper-bound on the Failure-In-Time rate. We assume that the occurrence of soft errors follows a Poisson process. Therefore within a time-memory extent T, the probability that k errors happen is:

$$Pr_{\lambda,T}[N = k] = \frac{e^{-\lambda T}(\lambda T)^k}{k!} \quad (1)$$

where λ is the average error rate (i.e., the error occurs λ times on average for every unit of time-memory extent). And particularly the probability for no error occurrence ($k = 0$) during a measurement over time-memory extent T is:

$$Pr_{\lambda,T}[\text{no error}] = e^{-\lambda T} \quad (2)$$

For a given T and the number of error occurrences k, let us call Λ a p-probability upper-bound of the average error occurrence rate if:

$$\forall \lambda > \Lambda : \quad Pr_{\lambda,T}[N = k] < 1 - p$$

In other words, if a computing environment has an average error occurrence rate that is more than the p-probability upper-bound, then the chance for k error occurrence during a measurement of time-memory extent T is always less than $1 - p$.

We apply the above analysis and metric definition on the error measurement results of our deployed measurements. We first look at UR desktop measurement in which no error is reported. According to Equation (2), we know that $\frac{1}{T} \ln \frac{1}{1-p}$ is a p-probability upper-bound of the average error occurrence rate. Consequently, since $T = 428$ GB $\times$ day for the UR desktop measurement, we can calculate that 54.73 FIT per Mbit is a 99%-probability upper-bound of the average error occurrence rate for this environment.

We then examine the Ask.com environment excluding 9 servers with hard errors. In this environment, 2 (or fewer) soft errors over $T = 73,571$ GB $\times$ day yields a 99%-probability upper-bound of the average error occurrence rate at 0.56 FIT per Mbit. This is much lower than previously-reported error rate (200–5000 FIT per Mbit) that we summarized in Section 1.

4 Conclusion and Discussions

Our preliminary result suggests that the memory soft error rate in two real production systems (a rack-mounted

server environment and a desktop PC environment) is much lower than what the previous studies concluded. Particularly in the server environment, with high probability, the soft error rate is at least two orders of magnitude lower than those reported previously. We discuss several potential causes for this result.

- *Hardware layout*: In the IBM Blue Spruce experiment, O'Gorman et al. [12] suggested that the main source of cosmic ray comes from straight above. The Ask.com machines are arranged in a way such that memory DIMMs are plugged perpendicular to the horizontal plane. This could significantly reduce the area facing the particle bombardment.
- *Chip size reduction*: Given the continuous scaling of the VLSI technology, the size of a memory cell has reduced dramatically over the years. As a result, for an equal-capacity comparison, the probability of a particle hitting any cell in today's memory is much lower than that of a decade ago. We believe that this probability reduction outweighs the increased vulnerability of each cell due to the reduction in device critical charge.
- *DRAM vs. SRAM*: Measurements described in this paper only target the main memory, which is usually DRAM. Previous studies [7, 17] show that DRAM is less sensitive to cosmic rays than SRAM (typically used for fast-access cache today).

An understanding on the memory soft error rate demystifies an important part of whole-system reliability in today's production computer systems. It also provides the basis for evaluating whether software-level countermeasures against memory soft errors are urgently needed. Our results are still preliminary and our measurements are ongoing. We hope to be able to draw more complete conclusions from future measurement results. Additionally, soft errors can occur on components other than memory, which may affect system reliability in different ways. In the future, we also plan to devise methodologies to measure soft errors in other computer system components such as CPU register, SRAM cache, and system bus.

Acknowledgments We would like to thank the two dozen or so people at the University of Rochester CS Department who donated their desktops for our memory error monitoring. We are also grateful to Tao Yang and Alex Wong at Ask.com who helped us in acquiring administrative access to Ask.com Internet servers. Finally, we would also like to thank the USENIX anonymous reviewers for their helpful comments that improved this paper.

References

[1] A. Acharya and S. Setia. Availability and utility of idle memory in workstation clusters. In *SIGMETRICS*, pages 35–46, 1999.

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *SOSP*, pages 43–56, 2001.

[3] EDAC project. http://bluesmoke.sourceforge.net.

[4] J. Cipar, M. D. Corner, and E. D. Berger. Transparent contribution of memory. In *USENIX*, 2006.

[5] C. da Lu and D. A. Reed. Assessing fault sensitivity in MPI applications. In *Supercomputing*, 2004.

[6] J. Douceur and W. Bolosky. Progress-based Regulation of Low-importance Processes. In *SOSP*, pages 247–260, Kiawah Island, SC, Dec. 1999.

[7] A. H. Johnston. Scaling and technology issues for soft error rates. In *4th Annual Research Conf. on Reliability*, 2000.

[8] H. Kobayashi, K. Shiraishi, H. Tsuchiya, H. Usuki, Y. Nagai, and K. Takahisa. Evaluation of LSI soft errors induced by terrestrial cosmic rays and alpha particles. Technical report, Sony Corporation and RCNP Osaka University, 2001.

[9] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Trans. on Electron Devices*, 26(1):2–9, 1979.

[10] A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. J. F. Lie, D. Mannaru, A. Riska, and D. S. Milojicic. Susceptibility of commodity systems and software to memory soft errors. *IEEE Trans. on Computers*, 53(12):1557–1568, 2004.

[11] E. Normand. Single event upset at ground level. *IEEE Trans. on Nuclear Science*, 43(6):2742–2750, 1996.

[12] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM J. of Research and Development*, 40(1):41–50, 1996.

[13] Tezzaron Semiconductor. Soft errors in electronic memory. *White paper*, 2004. http://www.tezzaron.com/about/papers/papers.html.

[14] J. F. Ziegler. Terrestrial cosmic rays. *IBM J. of Research and Development*, 40(1):19–39, 1996.

[15] J. F. Ziegler et al. IBM experiments in soft fails in computer electronics (1978–1994). *IBM J. of Research and Development*, 40(1):3–18, 1996.

[16] J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, T. J. O'Gorman, and J. M. Ross. Accelerated testing for cosmic soft-error rate. *IBM J. of Research and Development*, 40(1):51–72, 1996.

[17] J. F. Ziegler, M. E. Nelson, J. D. Shell, R. J. Peterson, C. J. Gelderloos, H. P. Muhlfeld, and C. J. Montrose. Cosmic ray soft error rates of 16-Mb DRAM memory chips. *IEEE J. of Solid-State Circuits*, 33(2), 1998.

Addressing Email Loss with SureMail: Measurement, Design, and Evaluation

Sharad Agarwal
Microsoft Research

Venkata N. Padmanabhan
Microsoft Research

Dilip A. Joseph
U.C. Berkeley

Abstract— **We consider the problem of *silent* email loss in the Internet, where neither the sender nor the intended recipient is notified of the loss. Our detailed measurement study over several months shows a silent email loss rate of 0.71% to 1.02%. The silent loss of an important email can impose a high cost on users. We further show that spam filtering can be the significant cause of silent email loss, but not the sole cause.**

***SureMail* augments the existing SMTP-based email infrastructure with a notification system to make intended recipients aware of email they are missing. A notification is a short, fixed-format fingerprint of an email, constructed so as to preserve sender and recipient privacy, and prevent spoofing by spammers. SureMail is designed to be usable immediately by users without requiring the cooperation of their email providers, so it leaves the existing email infrastructure (including anti-spam infrastructure) untouched and does not require a PKI for email users. It places minimal demands on users, by automating the tasks of generating, retrieving, and verifying notifications. It alerts users only when there is actual email loss. Our prototype implementation demonstrates the effectiveness of SureMail in notifying recipients upon email loss.**

I. Introduction

The Internet SMTP-based email system does not guarantee the timely or even eventual delivery of messages. Email can sometimes be delayed by hours or days, or even fail to be delivered to the recipient(s). Sometimes, the users are not even notified that their email was lost. Such *silent* email loss (i.e., the message is lost without a trace, not merely bounced back or misrouted to the junk mail folder), even if infrequent, imposes a high cost on users in terms of missed opportunities, lost productivity, or needless misunderstanding. Our SureMail system addresses this problem. Our targeting of silent loss is not fundamental. It is a trivial policy change to consider emails sent to the junk folder as lost email.

Recent measurement studies [15, 26] have reported email loss in the range of 0.5%-5%. We conducted a more thorough measurement study spanning months and find a *silent* loss rate of 0.71%-1.02%. While the lack of direct information from the email infrastructure makes it difficult to pin down the cause of email loss, we present evidence from one popular email service that points to spam filtering being the main cause.

From anecdotal evidence, we believe email loss also occurs elsewhere on the Internet, beyond the 22 domains in our experiment. Some users of EarthLink lost up to 90% of email silently in June 2006 [10]. AOL instructs users on what to do when email goes missing [3]. There are companies [8] that offer email monitoring services for businesses concerned about email loss.

Our measurement findings suggest that the existing SMTP-based email system works over 95% of the time. So our approach is to augment the existing system rather than replace it with a new one of uncertain reliability. SureMail augments the existing SMTP-based email delivery system with a separate *notification* mechanism, to notify intended recipients when they are missing email. By notifying the intended recipient rather than the sender, SureMail preserves the asynchronous operation of email, together with the privacy it provides. By having small, fixed-format notifications that are interpreted by email clients rather than being presented to users, we avoid the notification system from becoming a vehicle for malware such as spam and viruses as the current email system is.

Unlike some prior work, a key goal is for SureMail to be usable immediately by email users, without requiring cooperation from email providers. By not modifying the email infrastructure (including not altering spam filters), SureMail ensures against any disruption to email delivery that its installation might otherwise cause. Further, given its limited, albeit useful, functionality, a notification system would likely need less frequent upgrades (often disruptive) than a featureful email system.

We believe there is significant value in simply informing users that email to them was lost (i.e., is in neither their "inbox" nor "junk" folders). They can then contact the identified sender in a variety of ways to obtain the missing information (e.g., over email, different email accounts, phone, instant messaging).

We present two complementary approaches to delivering notifications: in-band delivery using email headers and out-of-band delivery using a web storage system. We have implemented both approaches and plan to make a SureMail add-in for Microsoft Outlook 2003 available. As in most P2P systems, both senders and receivers need to use it to benefit from SureMail. Our evaluation of the out-of-band approach shows that over 99.9976% of notifications are delivered successfully. We show that the incremental cost of SureMail is orders of magnitude lower than that of email, and thus we believe it is reasonable to deploy it for the added benefit of reliability.

We first proposed SureMail in a HotNets 2005 position paper[17]. Our design has since evolved considerably. The novel contributions presented here include:

- Measurement of email loss designed to avoid the shortcomings of prior studies.
- A redesign of SureMail to support in-band and/or out-of-band notifications, and to allow posting of notifications by legitimate first-time senders.

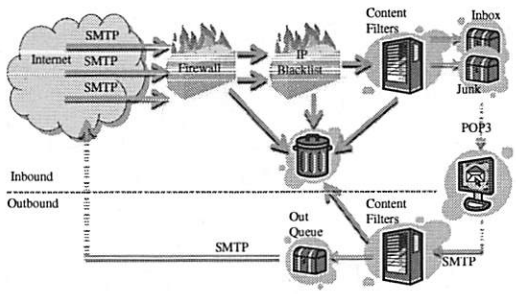


Fig. 1. A Typical Large Email Service Provider : top refers to inbound email, bottom to outbound email

- Implementation and evaluation of SureMail.

A. Outline

§ II describes a typical email service provider, various email filtering components and likely culprits of loss. This is followed by related work on email architectures and spam filter improvements. § III motivates the paper with our email loss measurement study, designed to avoid shortcomings of prior work [15, 26].

Given the significant amount of measured loss, we begin our design by first describing the *ideal* requirements of a solution in § IV. We strive to achieve these requirements in our design in § V. There are several challenges to be addressed within the security assumptions presented early in the section. § V-D presents a critical technique, *reply-based shared secret*, to prevent spammers from annoying users. § V-F explains how we distinguish some legitimate first-time senders from spammers.

SureMail allows notifications to be delivered in-band or out-of-band from email, or both in conjunction (§ VI). However, as with any new communication channel, the out-of-band technique introduces several challenges - our design is both low cost in terms of storage and message overhead, and resistant to security and privacy attacks. Our implementation and experimental evaluation of SureMail appear in § VII. We present a discussion of various issues pertaining to SureMail in § VIII.

II. Problem Background and Related Work

A. Typical Email Components and Email Loss

The typical email user may have a very simple view of their email system as that of their desktop client and the email server. In reality, large email providers tend to have complex architectures. Figure 1 provides a basic view of a typical provider. Each component in the figure may be replicated for load balancing, and some functionality may even be split among multiple devices.

The email system may be protected from malicious entities on the Internet by a firewall. It may block attacks, including excessive SMTP connections, SMTP connections with undesired options, connections with source addresses without reverse DNS entries, etc. The IP "blacklist" further blocks connections from certain IP addresses. This may include IP addresses of known spammers and open mail relays. Alternatively, a provider may use a "whitelist" policy where a source address may be automatically dropped if the volume of email sent by it falls below a threshold [4]. These connections are dropped without *any* inspection of the email content.

Any connections that pass through the firewall and IP blacklist are accepted and their email payloads are then processed by the content filters. These filters scan for viruses, worms and spam. Spam content can identified by a variety of techniques that involve comparing the text in the email to text from typical spam. Email that passes these filters will be stored in user inboxes. Email that is suspicious may be stored in junk mail folders instead. Email that is reliably identified as malware or spam may be thrown away without hitting any storage.

In the figure, only emails going to the "Inbox" or "Junk" folders are actually stored. Everything else is shown as going to the trash, which indicates that those emails are simply dropped. The major reason that not all emails are stored is a matter of volume. A major corporation's IT staff told us that about 90% of incoming email is dropped before it reaches user mail stores. Only the remaining 10% reach the inbox or junk folder.

Note that SMTP is not an end-to-end reliable protocol. Thus any of these components can temporarily fail due to overload, upgrade or maintenance and cause even more emails to be delayed or lost. Ever increasing volumes of spam and virus attacks make the infrastructure more susceptible to overload and failure.

In the outbound direction, email can also be lost. Emails composed by the user are typically sent via SMTP to the Internet. Some large email providers apply content filtering on outbound emails before they go to the Internet to filter out malware and spam that their users may be sending. This is to deter spammers who obtain email accounts on these providers in violation of the user agreement. Also, if too much malware or spam is sent by a particular provider, other providers may add that provider to their IP blacklist (or remove from their whitelist). Sometimes, email sent by travelers can be lost if they are forced to use a hotel's SMTP server that is not on the whitelists of destination SMTP servers.

Given such extensive filtering, it is not surprising that some legitimate email gets discarded entirely, not merely misrouted to the recipient's junk mail folder (we do *not* consider the latter as email loss).

SMTP allows a server to generate non-delivery messages ("bouncebacks") for emails it cannot deliver. This would only occur for emails where the incoming SMTP connection is accepted and parsed. In Figure 1, any drops by the firewall or IP blacklist would not cause any bouncebacks to be generated. For the emails dropped by

the content filters, several issues reduce the effectiveness of bouncebacks in making email loss non-silent: (i) Typical content spam filters do not generate bouncebacks. (ii) Bouncebacks may be sent to spoofed source addresses, leading to user apathy toward such messages, or worse, to their classification as spam. (iii) Bounceback generation may be disallowed for privacy (e.g., to avoid leaking information on the (in)validity of email addresses). (iv) Servers sometimes (e.g., after a disk crash) do not have enough information to generate bouncebacks. (v) Bouncebacks cannot warn users about emails lost from a email server to a client.

B. Prior Work on Email Unreliability

We are aware of two recent measurement studies of silent email loss. Afergan et al. [15] measured silent email loss by recording the absence of bouncebacks for emails sent to non-existent addresses. 60 out of the 1468 servers measured exhibited a silent email loss rate of over 5%, with several others with more modest but still non-negligible loss rates of 0.1-5%. However, a shortcoming of their methodology is that bouncebacks may not reflect the true health of the email system for normal emails and many domains do not generate bouncebacks.

Lang [26] used a more direct methodology to measure email delays and losses. 40 email accounts across 16 domains received emails over a 3-month period. Their overall silent email loss rate is 0.69%, with it being over 4% in some cases. While that study does not depend on bouncebacks, it may be biased by the use of a single sender for all emails and the use of very atypical emails (no email body; subject is a message sequence number) that could significantly bias these being filtered as spam. Our study addresses some of these shortcomings.

To put these findings in perspective, even a silent loss rate around 0.5% (1 lost email among 200 sent, on average) would be a serious problem, especially since a user has little control over which emails are lost.

Prior proposals to address the email unreliability problem range from augmentating the current email system to radical redesign. The message disposition notification mechanism [21] (i.e. "read receipts"), enables senders to request that the recipient send an acknowledgment when an email has been downloaded or read. We believe that most users do not enable this feature in their email clients as it exposes too much private information — when the user reads reads email, conflicting with the inherent "asynchronous" use of email. Read receipts also enable spammers to detect active email accounts.

While re-architecting the email delivery system to enhance reliability (e.g., POST [5, 27]) is certainly desirable, that alone will not solve the problem because of loss due to spam filters. Although a public key infrastructure (PKI) for users, as assumed by POST, can help with the spam problem, it can be an impediment for deployment. In contrast, SureMail does not modify the underlying email delivery system and keeps the notification layer separate. This avoids the need to build (or modify) the complex functionality of an email delivery system and ensures that even in the worst case, SureMail does not adversely affect email delivery.

C. Spam Filters and Whitelisting

Improved spam filtering techniques ([9, 11, 13]), reduce false positives while still doing effective filtering. However, it is difficult to entirely eliminate false positives as spam constantly evolves to mimic legitimate traffic. Very high spam volumes often necessitate content-independent filtering (e.g., IP blacklisting) to reduce processing load on email servers.

Whitelisting email senders (using social relationships or otherwise [19, 23]) to bypass spam filters is complementary to SureMail. SureMail tries to notify recipients upon email loss, regardless of the cause, without actually preventing the loss. Whitelisting seeks to prevent email loss specifically due to spam filtering. Thus it needs to operate on email before it hits the spam filtering infrastructure. This requires the cooperation of the email administrators and convincing them that this modification to their servers will not negatively impact email delivery. In contrast, SureMail leaves the email infrastructure untouched, and allows individual users to start using the system without involving their email administrators. Finally, if a trusted sender's computer is compromised, potentially harmful emails may be whitelisted through the filtering infrastructure. In SureMail, this compromise only results in bogus notifications being delivered. We rely on human involvement for conveying the missing information, because email loss is relatively rare.

III. Email Loss Measurement

We begin by quantifying the extent of email loss in the existing email system. Due to privacy issues and the difficulties of monitoring disparate email servers, we resort to a controlled experiment where we send all the email, like [26]. However, we improve on their study by using multiple sending accounts, more realistic email content, and shedding light on the causes of email loss.

A. Experiment Setup

1) Email Accounts: To measure email loss on the Internet, we obtained email accounts on several academic, commercial and corporate domains (see Table IV). The non-academic domains include free email providers, ones that charge us for POP or IMAP access, and a private corporation. The domains are spread across Australia, Canada, New Zealand, UK and USA. In most cases, we obtained two mailboxes to catch cases where accounts on the same domain are configured differently or map

1) Seed random number generator
2) Pick a sender email address at random
3) Pick a receiver email address at random
4) Pick an email from corpus at random
5) Parse email and use the subject and body
6) With 30% probability, add an attachment, selected at random
7) If such an email (sender, receiver, subject, body, attachment) has not been sent before, send
8) Log sent email and any SMTP error codes
9) Sleep for a random period under a few seconds
10) Go back to step 2

Fig. 2. Pseudo-code for Sending Process

to different servers. Most systems allowed us to retrieve emails over POP3, IMAP, Microsoft Exchange, or RPC over HTTP. Many allowed us to programmatically send emails using SMTP. Overall, we have 46 email accounts: 44 allow receiving email, and 38 allow sending.

2) Email Content: We programmatically send and receive emails across these 46 accounts. To mimic content sent by real legitimate users, we use the "Enron corpus", a large set of emails made public during the legal proceedings against the Enron corporation. We obtained a subset of this corpus [12] containing about 1700 messages manually selected for business-related content, while avoiding spam. Of these, we use a subset of 1266 emails with unique subjects, which facilitates the subsequent matching of sent emails with received emails. We use only the body and subject from the corpus and ignore the rest of the header.

We do not use any attachments from the corpus for fear that sending malware might bias our findings. To understand the impact, if any, of attachments on email loss, we picked 16 files of 7 different formats and various content : marketing, technical, and humorous materials (see Table III). The largest is about 105 KB since we do not want to overburden the hosting email domains. We did not include executables and scripts, since they increase loss due to virus and trojan scanners – at least one of the domains drops all emails with executable attachments. We do not attempt an exhaustive study of email loss due to attachments, but instead estimate if typical attachments influence the observed loss rate.

3) System Setup: We use the sending process in Figure 2. It is codified in a Perl program which uses separate C programs that handle SMTP connections for sending emails. We bias the sleep period in step 9 to not violate the daily volume limits in most account agreements. While most accounts have very similar limits, the msn.com and microsoft.com accounts allow us to send and retrieve almost 10 times more emails. Thus steps 2 and 3 are appropriately biased to more frequently send to and from these 6 accounts.

To retrieve emails, we configured Mozilla Thunderbird 1.5 to download emails from all receiving accounts. We download emails from the inbox of each account, as well as any junk mail or spam folders. Whenever allowed, we configured the accounts to disable junk mail filtering and/or created a whitelist with all 46 account addresses. We use Windows XP SP2 machines located on Microsoft's network to send and retrieve email.

Once an experiment has completed running, we use a Perl program to parse the sending logs and feed them into a Microsoft SQL Server 2005 database. A second Perl program parses the Thunderbird mail files and feeds the retrieved emails into the same database. The program also attempts to parse the contents of any bouncebacks to determine which original email bounced. However, in some cases, not enough information is present in the bounceback to uniquely identify the lost email or the format of the bounceback is atypical and difficult to parse. We issue SQL queries to match sent emails with received emails, and calculate email loss statistics. The matching is done based on the following fields: sender email address, receiver email address, subject, attachment name. We do not use the body of the email for matching, because some email providers (e.g. Yahoo) insert advertisements. Hence our corpus consists of emails with unique subjects.

B. Email Loss Findings

We conducted three separate email loss experiments, summarized in Table I. The setup for experiments #1 and #2 is slightly different and is described in a technical report [16]. In this paper, we focus on the latest study, #3. We received more emails than we sent, primarily due to spam and domain announcements. Our SQL queries for matching sent emails with received emails ignore these extraneous emails. The overall loss rate is about 1.79%. We received about 1216 bouncebacks, with various status codes and reasons, not all of which we could accurately match to the sent email. 10 pairs of senders and receivers were unable to exchange any email during our experiment. These constitute the 363 "hard failures". If we remove these unusual hard failures, and count all bouncebacks as successful notifications of delay or loss, we have a conservative *silent* loss rate of 1.02%. It is difficult to pin down the exact cause of each loss due to the opacity of so many email domains. However, we later attempt to identify the possible causes.

The silent loss rate appears to have increased slightly over time across the three experiments. We speculate that spam filters have had to more aggressively adapt to increasing volumes of spam. In private communication, two of the email providers confirmed that they updated spam filters almost continuously during this period.

Our experiment was biased toward sending and receiving more emails via the msn.com and microsoft.com accounts due to the higher allowed limits. If we remove these 6 accounts completely from our analysis, we have the results shown in Table II. This overall loss rate of

	Exp. 1	Exp. 2	Exp. 3
Sending accounts	36	36	38
Receiving accounts	42	42	44
Emails in corpus	1266	1266	1266
Attachment probability	0.3	0.3	0.3
Start date	11/18/05	01/11/06	09/06/06
End date	01/11/06	02/08/06	10/04/06
Days	54	29	29
Emails sent	138944	19435	203454
Emails received	144949	21015	213043
Emails lost	2530	653	3648
Total loss rate (lost/sent)	1.82%	3.36%	1.79%
Bouncebacks received	982	406	1216
Hard failures	565	70	363
Conservative silent loss rate	0.71%	0.91%	1.02%

TABLE I. Email Loss Statistics

Emails sent	88711
Emails lost	1653
Total loss rate	1653/88711 = 1.86%

TABLE II. Loss Statistics w/o msn.com and microsoft.com

1.86% is very similar to the 1.79% rate from Table I. So we believe our findings are not biased by the higher sending and receiving rates for these 6 accounts.

C. Detailed Findings

We now present detailed loss statistics for experiment 3, broken down by attachment, email body and email account. We only consider overall loss rates since matching bouncebacks to the specific email sent is difficult.

1) Loss by Attachment: Table III presents the loss by attachment. We want to estimate if the type of attachment or its content dramatically influences loss. We had 16 attachments of 7 types. For instance, we had 2 GIFs – home_main.gif and phd050305s.gif – and emails that included them suffered loss rates of 1.68% and 1.99%, respectively. While we do not observe a significant deviation from the overall loss of 1.79%, HTML attachments did suffer higher loss. We speculate that since HTML is becoming a more popular email body format, content-based spam filters are more actively parsing HTML.

2) Loss by Email Subject / Body: Figure 3 plots the loss rates, sorted from high to low, for our 1266 distinct email subjects/bodies. We see that some emails have significantly higher loss rates than others. Most of the email bodies with loss rates above 10% appear to contain business proposals, Enron-related news, and stock details. Even if we ignore these email bodies, most of the rest of the corpus does have a non-negligible loss rate. Thus, our findings are not a result of a few "bad apples" in our corpus, but due to a more general problem.

3) Loss by Account: Table IV shows the total loss rates for each account on each domain. Column 4 lists the aggregate number of emails sent to each account from all of the 38 sending accounts. Column 4 lists the loss rates experienced by these set of emails. Column 5 presents the aggregate number of emails sent from each of these accounts to all of the 44 receiving accounts. The last

Attachment Type	Emails Sent	Loss %
(none)	133198	1.56
2 * JPEG	2322, 4656	1.55, 2.10
2 * GIF	4597, 4532	1.68, 1.99
3 * HTML	4808, 4733, 4768	4.53, 3.53, 4.28
3 * MS DOC	4658, 4582, 4716	2.36, 1.88, 1.68
2 * MS PPT	4596, 2363	1.59, 1.78
2 * PDF	4670, 4808	1.56, 1.71
2 * ZIP	4749, 4698	1.41, 1.43

TABLE III. Email Loss Statistics by Attachment

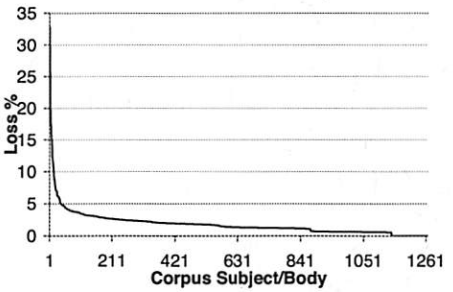


Fig. 3. Loss Statistics by Email Subject and Body

column is the loss experienced by these emails. We make two observations here. First, our overall loss rates are not influenced by a few bad domains or accounts — although a few accounts experienced no email loss, there is loss across most domains and accounts. Second, within each domain, both accounts tend to suffer similar loss rates [1].

4) Cause of Email Loss: In general, it is difficult to exactly determine the cause of each instance of loss because of the complexities of the myriad of email systems and our lack of access to the innards of these systems. We speculate that the likely causes of loss are aggressive spam filters and errors in the forwarding or storage software and hardware.

We focus here on the 4 msn.com accounts, for which we obtained special access. We disabled the content filters (see Figure 1) for *c@msn.com* and *d@msn.com*. Any incoming emails that the content filters would have thrown away are "tagged", and the number of such emails are shown in column 4 of Table V. The *a@msn.com* and *b@msn.com* accounts are regular accounts [2]. The "Loss %" column shows the overall loss rate, while the last column shows what the rate would have been had the "tagged" emails been lost. The regular loss rates for c and d are relatively small, but the "tagged loss" rates for c and d are similar to the regular loss rate for a and b. So we hypothesize that content-based spam filters are the main cause for email loss for msn.com. However, they are not the sole cause, as indicated by the non-zero "Loss %" column for c and d. Due to this and the lack of perfect

[1]The second account at fusemail.com suffered a high receive loss rate. No emails were delivered between 09/30/2006 and 10/02/2006. Technical support was unable to determine the cause of the loss. We suspect server failure or loss between the server and our client.

[2]Note that we still treat any emails sent to the "Junk" folder for any account as though they were delivered to the Inbox and not lost.

Domain	T	Sent to		Recvr loss%		Sent from		Sendr loss %	
		A	B	A	B	A	B	A	B
aim.com	I	2306	2370	3.82	4.35	568	174	0.70	0.00
bluebottle.com	P	2336	2454	0.04	0.08	2446	2452	1.59	1.39
cs.columbia.edu	I	2338	2387	7.83	8.09				
cs.princeton.edu	P	2416	2341	0.29	0.04				
cs.ucla.edu	P	2346	2320	0.94	1.03				
cs.utexas.edu	P	2387	2395	1.01	1.29	3669	3628	2.89	3.03
cs.wisc.edu	I	2386	2350	0.54	0.21	3754	3592	2.08	2.31
cubinlab.ee.mu.oz.au	P	2341	2408	0.04	0.08				
eecs.berkeley.edu	I	1193	2321	4.36	3.62	1893	3688	2.11	2.03
fusemail.com	I	2425	2350	0.00	20.26	3740	3525	2.65	1.53
gawab.com	P					3652	3666	9.94	9.36
gmail.com	P	2313	2369	3.59	3.29	3680	3595	2.20	1.81
microsoft.com	E	19330	18504	1.77	1.62	35079	35157	0.62	0.65
msn.com	H	19465	18932	3.25	3.02	6757	6775	0.92	1.05
msn.com	H	19122	19390	0.37	0.41	6807	6793	1.23	1.06
nerdshack.com	P	2387	2305	0.00	0.17	3651	3588	4.14	4.26
nms.lcs.mit.edu	P	2421	2389	0.00	0.00	3649	3657	1.12	1.12
ulmail.net	P	2378	2360	0.00	0.00	3715	3697	5.49	5.63
usc.edu	P	2314	2371	0.04	0.00	3608	3731	1.36	1.07
yahoo.com	P	2324	2355	2.54	1.10	3835	3630	0.81	0.94
yahoo.co.uk	P	2442	2323	0.00	0.00	3474	3577	1.07	1.06
cs.uwaterloo.ca	P	2464	2323	1.87	1.89	3667	3602	1.77	1.30

TABLE IV. Loss by Email Account; numbers missing where programmatic sending/retrieving not allowed; T=Protocol type: E:Exchange,H:RPC/HTTP,I:IMAP,P:POP3; A=1st account, B=2nd

Receiver	Sent	Matched	Tagged	Loss %	Tagged loss %
a@msn.com	19465	18833		3.25	
b@msn.com	18932	18361		3.02	
c@msn.com	19122	19052	503	0.37	3.00
d@msn.com	19390	19311	531	0.41	3.15

TABLE V. Loss Statistics to msn.com

spam filters, there is value in email loss notification.

To summarize our experiment, we found significant total loss rates of 1.79% to 3.36%, with silent loss rates of 0.71% to 1.02%. That is, if a user sends about 30 emails a day, over the course of a year, over 3 days worth of emails get silently dropped. Based on our detailed results, we believe our findings were not biased significantly by the choice of attachments, message bodies and subjects, email domains, and individual accounts within the domains. Thus we believe that a system for addressing lost email will be of significant benefit to users. Our measured loss may not correspond exactly to the typical user experience, because:

– The 46 accounts on 22 domains in our experiment represent a small fraction of the worldwide email system.

– Our mix of intra- and inter-domain emails, the sending and receiving rates per account, and even the content (despite being derived from a real corpus), may not match user workload.

– In our experiments, all email addresses had previously emailed each other, and thus none was a "first-time sender" to any. First-time senders may experience higher loss, but despite that we found significant loss.

– A user may not be aware that their email was lost. Our experiment avoids this uncertainty by controlling both the senders and the receivers.

IV. Design Requirements

We believe the following properties of a solution to email loss will lead to rapid adoption and deployment.

1) **Cause minimal disruption:** Rather than replace the current system, which works for the vast majority of email, with a new system of uncertain reliability, augment it to improve reliability. It should inter-operate seamlessly with the existing email infrastructure (unmodified servers, mail relays, etc.), with additions restricted to software running outside it (e.g. on end-hosts). Users should benefit from the system without requiring cooperation from their email domain administrators.

2) **Place minimal demands on the user:** Ideally, user interaction should be limited only to actual instances of email loss; otherwise, he/she should not be involved any more than in the current email system.

3) **Preserve asynchronous operation:** Email maintains a loose coupling between senders and recipients, providing a useful "social cushion" between them. The sender does not know whether or when an email is downloaded or read. Recipients do not know whether a sender is "online". Such asynchronous operation should be preserved, unlike in other forms of communication such as telephony, IM, and email "read receipts".

4) **Preserve privacy:** The solution should not reveal any more about a user's email communication behavior than the current system does. For instance, it should not be possible for a user to determine the volume or content of emails sent/received by another user, the recipients/senders of those emails, how often that user checks email, etc. However, as it stands today, email is vulnerable to snooping, whether on the wire or at the servers. We do *not* seek to rectify this issue.

5) **Preserve repudiability:** Repudiability is a key element of email and other forms of casual communication such as IM [14, 18]. In the current email infrastructure, a receiver can identify the sender from the header, but cannot *prove* the authorship of the email to a third-party, unless the sender chose to digitally sign it. Any solution to email loss should not *force* senders to sign emails or facilitate receivers in proving authorship. As an analogy, people are often more comfortable identifying themselves and communicating sensitive information in person than in written communication, since the latter leaves a paper trail with proof of authorship to a third party. Note that PKI or PGP based authentication of email users, is unsuitable from the viewpoint of providing repudiability.

6) **Maintain defenses against spam and viruses:** It should be no easier for spam or viruses to circumvent existing defenses or tell if an email address is valid.

7) **Minimize overhead:** The solution should minimize network and compute overheads from additional messaging (e.g. sending *all* emails twice would significantly overload some email servers and network pipes).

V. SureMail Design

SureMail is designed to satisfy the requirements listed above. We continue to use the current email system for message delivery, but augment it with a separate, low-cost *notification* system. When a client sends an email,

it also sends a notification, which is a short, fixed-format fingerprint of the email. We consider various notification delivery vehicles in Section VI, including email headers and a web service. If email loss occurs, the receiving client gets a notification for the email but not the email itself. After waiting long enough for missing email to appear (Section VIII-A), it alerts the recipient user, informing him/her about email loss from the sending user specified in the notification. SureMail does not dictate what action the recipient user should take at this point. The user may ignore it or contact the sender via email, IM, or phone, presenting the fingerprint in the notification to help the sender find the lost email.

The bulk of the work (creating, posting, checking, and retrieving notifications) is done automatically by the SureMail software. The user is involved only when a lost email is detected. SureMail allows a user to determine if he/she is missing any emails sent to him/her. It does *not* notify the sender about the status of email delivery. Thus SureMail assures senders that either email is delivered or the intended recipients will discover that it is missing. While the concept is simple, there are several key challenges that SureMail must address:

1) Prevent notifications themselves from being used as a vehicle for spam or malware: We use very short, 64-byte, fixed-format notifications (Figure 5), which are interpreted by the SureMail client rather than being presented directly to the user. Section V-C has the details.

2) Avoid the need to apply spam filters on notifications: Unlike email spam, notification spam does not directly benefit the spammer because of the restrictive nature of notifications noted above. However, to block bogus notifications, which could annoy users (e.g., by alerting them to the "loss" of non-existent email or spam), we present our reply-based shared secret technique in Section V-D.

3) Prevent or minimize information leakage from notifications: Information such as the fact that a particular email address is active or that user x has emailed user y could be sensitive even if the email content itself is not revealed. Section VI addresses these problems.

A. Security Assumptions

1) We assume that when a recipient chooses to reply to an email, they are implicitly indicating that the sender of the email is legitimate. I.e., users are very unlikely to reply to spam. We consider this unlikely possibility of a user being tricked into replying to spam in Section V-E.

2) We assume that the attacker cannot mount a man-in-the-middle attack (intercept and modify emails exchanged by a user). While we do not rule out the possibility of eavesdropping, we believe that in practice even this would be hard: an attacker would require access to the path of (remote) users' email. Furthermore, if an attacker does gain access to user email, that compromises user privacy more directly than subverting notifications.

3) If a separate infrastructure is used to deliver notifications, it may not be entirely trustworthy. The notification infrastructure may *try* to spy on users' email activity (e.g., who is emailing whom) or generate bogus notifications. Bogus notifications are a more serious problem than dropped notifications (which the notification infrastructure can always cause), since the former imposes a cognitive load on users while the latter leaves users no worse off than with the current email system.

B. Notation

Unless otherwise stated, S and R represent the sender and the recipient clients (and users) of an email. We assume that all nodes agree on: **(i)** a cryptographic hash function $H(X)$ that operates on string X to produce a short, fixed-length digest (e.g., a 20-byte digest with SHA1), and is pre-image and collision resistant; **(ii)** a message authentication code (i.e., a keyed hash function), $MAC_k(X)$, that operates on string X and key k to produce a short, fixed-length digest (e.g., a 20-byte digest with HMAC-SHA1). A MAC can be used with various well-known keys (e.g., k') to generate new hash functions (e.g., $H'(X) = MAC_{k'}(X)$); **(iii)** a symmetric encryption function, $E_k(X)$; **(iv)** a digital signature scheme to produce a signed message $SIG_k[X]$ using private key k.

C. Notification Basics

A notification is a short, 64-byte, fixed-format structure interpreted by software (not read by humans). This is in contrast to rich and extensible email (e.g. attachments, embedded HTML) that is often orders of magnitude larger. Consequently, it is hard for an attacker to send malware or spam in notifications. This makes it easier to reliably deliver notifications compared to email.

To identify an email in its notification, we cannot use the `Message-ID` field contained in some email headers because it may be set or reset beyond the sending client (e.g., by a Microsoft Exchange server), making it inaccessible to the sender. So we embed a new `X-SureMailID` header with a 20-byte SureMail message ID ($smID$), which is unique with high likelihood.

D. Reply-based Shared Secret

A notification also needs to identify the sender S, so that R knows who to contact for the missing email. However, we cannot simply insert S's email address into the notification since that can be easily spoofed. We instead use our reply-based shared secret scheme which blocks spoofed or bogus notifications, protects the identity of S, and needs no user involvement. It automatically establishes a shared secret between two users who have emailed each other before. We consider the first-time sender (FTS) case in Section V-F.

Say S sends an email M_1 to R and receives a reply M_1' from R. The SureMail client software at S and R

uses the corresponding SureMail message IDs, $smID_{M_1}$ and $smID_{M_1'}$, to perform a Diffie-Hellman exchange and set up a shared secret, $smSS_1$, known only to S and R. $smSS_1$ is computed and remembered by R's SureMail client automatically when user R replies to the email (recall assumption 1 from Section V-A) and by S's client when it receives the reply. S can then use $smSS_1$ to authenticate notifications it posts to R and to securely convey its identity to R (and R alone). Note that R would have to establish a separate shared secret with S, based on an email exchange in the opposite direction, for notifications that R posts to S. The notification for a new message, M_{new} from S to R is constructed as follows:

$$N = \{T, H(smID_{M_{new}}), H(smSS_1), MAC_{smSS_1}(T, H(smID_{M_{new}}))\}$$

T is a timestamp. $H(smID_{M_{new}})$ identifies the new message. $H(smSS_1)$ implicitly identifies S to R, and R alone. $MAC_{smSS_1}(T, H(smID_{M_{new}}))$ proves to R that this is a genuine notification from S with an untampered timestamp, since only S and R know $smSS_1$.

Upon retrieving a notification, the SureMail client at R checks to see if it is recent (based on T) and genuine (i.e., corresponds to a known shared secret). If it is and the corresponding email is not received soon enough (see Section VIII-A), it alerts user R and presents S's email address. Old and invalid notifications are ignored.

When an email is sent to multiple recipients, a separate notification is constructed and posted for each using the respective reply-based shared secrets.

1) Shared Secret Maintenance: The SureMail clients at S and R perform a Diffie-Hellman exchange to establish a shared secret only if each was the sole addressee in an email exchange. So emails sent to multiple recipients and those send to mailing lists are excluded.

Each node remembers two sets of shared secrets for each correspondent: one for posting notifications and the other for validating received notifications. These are updated with each new email exchange. S remembers the smIDs of all messages sent by it since the most recent one replied to by R. Likewise, R remembers the smIDs of all emails from S that it had replied to (or just the smSSs derived from such emails), since the most recent one that S used as a shared secret in a notification.[3]

This constant renewal allows the shared secret to be reestablished if the user starts afresh, say after a disk crash. It also helps purge a bogus shared secret, such as when a spammer tricks R into responding to a forged email spoofed to be from S (see Section V-E).

2) Reply Detection: To help a client determine that an incoming email is a reply to a prior outgoing email, we include an `X-SureMailInReplyTo` (smIRT) header to echo the smID of the original email. While similar, we cannot use the "In-Reply-To" header included by most email clients because the message ID of the original email may not be available (see Section V-C).

[3] R could instead remember a few older smSSs as well to accommodate the possibility of an old notification, constructed using an older smSS, being reordered and delivered late. Given the (slow) human timescale of the email-reply cycle that generates a new smSS, remembering just a few recent smSSs should be sufficient.

E. Security Properties

The Diffie-Hellman exchange ensures that the shared secret, $smSS$, between S and R is not known to an attacker A that eavesdrops on the email exchange or on the notification. A cannot learn who posted a notification (thereby preserving privacy) or post fake notifications deemed as valid.

However, consider a more difficult attack. A sends R a spoofed email that appears to be from S and is also realistic enough that user R is tricked into replying, thus making R remember a bogus shared secret. If A also eavesdrops on the reply, it can learn this shared secret and then post bogus notifications. However, even if A manages to pull off this attack once, the bogus shared secret gets flushed out with renewals (Section V-D.1).

Unlike PKI/PGP-based systems, our reply-based shared secret scheme does not require any human involvement to set up keys. Our system also preserves the repudiability of email (see Section IV). The shared secret between S and R is not meaningful to a third party. So although R can satisfy itself with the authenticity of N from S, it cannot use N to prove to a third party that M_{new} was sent by S. In contrast, if SureMail required PKI/PGP, M_{new}'s author could be proved to anyone.

F. First-time Sender (FTS)

While the experiment in Section III showed email loss in the non-FTS case, it is desirable to address the FTS case as well. In our reply-based shared secret scheme, a legitimate FTS, who has not exchanged email with the recipient previously, cannot construct an authentic notification. Although it might seem that a legitimate FTS is indistinguishable from a spammer, in practice, there are social or business [4] links that may set apart the legitimate FTS from a spammer. [20] shows that email networks exhibit *small-world* properties and RE: [23] leverages it to create effective whitelists. So, although the FTS F may never have communicated with the intended recipient R, it is likely that F has communicated with an intermediary I who has in turn communicated with R. For example, I may be a colleague at R's institution. I is thus in a position to "introduce" F to R.

We want F to be able post a notification that R would treat as authentic. In leveraging F and R's relationship with I, we seek to preserve privacy by preventing: **(a)** I from learning that F intends to communicate with

[4] Enterprise networks typically have authentication mechanisms such as Kerberos that can validate legitimate employees.

R, **(b)** F from learning that I and R have previously communicated, and **(c)** R from learning that F and I have previously communicated. In the event that (c) cannot be satisfied, we consider a diluted goal, **(c')**, which is to prevent R from learning about any of F's correspondents other than those it shares in common with F.

There has been prior work on similar problems in the context of exchanging email whitelists. In LOAF [19], users exchange address book information with their correspondents using Bloom filters. This scheme satisfies property (a), but not (b), (c) or (c'). RE: [23] uses a novel friend-of-friend (FoF) approach. Using a homomorphic encryption-based private matching protocol [22], RE: ensures properties (a), (b), and (c'), but not (c). (However, RE: may permit a malicious R to violate (c') and learn about friends of F that it doesn't share in common, per Section 6 of [23]). Our introduction mechanism also satisfies (a), (b), and (c'), but not (c).

1) SureMail Introduction Mechanism: In our introduction mechanism, every node I generates a public-private key pair, (S_I, P_I). It shares the public ("shared") key, S_I, with its correspondents to establish a *common* secret known only to its correspondents (the public key is *not* shared with others — there is no PKI). In addition, I generates a public-private key pair, (S_{IF}, P_{IF}), for each new correspondent, say F, which it hands to F along with a signed token containing the F's email address and the newly generated public key, S_{IF}. So when it becomes a correspondent of I, F learns the common secret S_I, the key pair (S_{IF}, P_{IF}) generated for it, and the token $X_{SF} = SIG_{P_I}[F, S_{IF}]$ signed with I's private key, P_I.

F can use S_I and X_{SF} to authenticate notifications it posts for R, which is another correspondent of I. A notification from F for an email M sent to R is :

$N_{FTS} = \{T, H(S_I), E_{H'(S_I)}(T, X_{SF}, R),$
$SIG_{P_{IF}}[H(smID_M)], SIG_{P_{IF}}[E_{H'(S_I)}(T, X_{SF}, R)]\}$

$H(S_I)$ allows R to look up S_I from its store of common secrets obtained from its correspondents (and discard the notification if the lookup fails). It can then compute the key $H'(S_I)$ and decrypt $E_{H'(S_I)}(T, X_{SF}, R)$. Note that unlike in the non-FTS construction from Section V-D, F and R need to be identified explicitly since there is no pairwise shared context between them. Also, the signed token X_{SF} prevents F from assuming an arbitrary identity, which helps defend against certain attacks (see the security properties discussion in Section V-F.3).

After verifying the signed token X_{SF}, R uses F's public key to validate $SIG_{P_{IF}}[H(smID_M)]$, preventing an attacker from tampering with the message ID. The encrypted token, $E_{H'(S_I)}(T, X_{SF}, R)$, is also signed, which among other things ties the notification down to recipient R. This prevents R from turning around and reusing it in a fake notification purporting to be from F and destined to another correspondent of I, say Z.

If the introduction is deemed as valid per the above procedure, R honors the notification. Otherwise, R ignores it. Of course, as a side-effect of processing a valid notification, R also learns that both F and itself share I as a correspondent, which violates property (c) noted above but is consistent with (c').

2) Picking an Intermediary: F needs to pick an intermediary I whose shared secret (S_I) it should use for the introduction. We believe that it is appropriate to rely on human input, since the FTS scenario would occur relatively infrequently and the user at F is in the best position to decide which I would likely have communicated with R and whose identity it is allowed to leak to R. So when the SureMail client at F detects that it is an FTS with respect to R, it prompts the user for a recommendation of one or more intermediaries from among F's correspondents. The client aids the process by automatically listing correspondents who are in the same email domain as R and hence are likely to be suitable intermediaries. If the user picks more than one intermediary, a separate notification (constructed as N_{FTS} above) would be posted corresponding to each.

As an alternative, we can avoid user involvement by having F's client automatically post multiple notifications constructed with the shared secrets obtained from each of its correspondents. R could then determine if any match with its list of correspondents, and if so, deem the introduction as valid. Since the shared secrets are opaque tokens, R does not learn anything from the shared secrets that originated from correspondents of F that are not common to R. Thus property (c') is satisfied. However, note that this extreme procedure generates a notification traffic volume proportional to the number of correspondents, as in schemes such as RE:.

3) Security Properties: The notification construction prevents an attacker (other than a correspondent of I), who sees N_{FTS}, from learning the secret S_I, or identifying F or R. Furthermore, even a correspondent of I is prevented from constructing a fake notifications purporting to be from F. Also, note that repudiability is also preserved since F only signs the message ID using P_{IF}, not the message content itself.

In terms of privacy, property (a) is satisfied since I is oblivious to the communication between F and R. Property (c') is also satisfied as noted above. While property (b) is also satisfied by the protocol as described, there is the possibility of a subtle social engineering attack: F could post a notification for R (with I as the intermediary) but *not* send the corresponding email. If (an anxious) R inquires with F about the "missing" email, F can conclude that I and R must be correspondents (for otherwise R would have just ignored the notification).

However, we believe that this social engineering attack

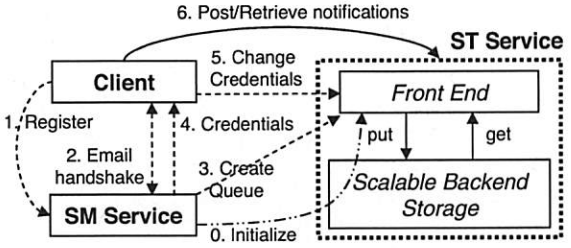


Fig. 4. Interaction between a client, SM, and ST. The various operations are numbered in the order in which they are invoked. The dashed-dotted line operation (#0) is invoked at system initialization time. The dashed line operations (#1-#5) are invoked at the time of client registration. The solid line operation (#6) is invoked during normal operation, i.e., posting and retrieval of notifications.

would be hard to employ in practice. Note that for the attack to succeed, F would have to be in a position to receive R's inquiry about the missing email. However, F cannot use a fake email identity, say F', since it wouldn't have the corresponding $X_{IF'}$ signed by I. So F would likely have to identify itself and run the risk of being caught if it repeats the trick over time.

VI. Delivering SureMail Notifications

We now present two complementary approaches to delivering notifications: in-band and out-of-band.

A. In-band Notifications

In-band notifications are embedded within emails themselves — the notification for an email is embedded later emails between the same sender-recipient pair. Suppose S sends R three emails: M_1, M_2, M_3. M_2 will contain the notification for M_1. M_3 will contain the notifications for both M_1 and M_2. If R did not receive M_1, it will find out when it receives either M_2 or M_3. This is akin to how TCP detects a missing packet via a sequence "hole". Since these notifications are confined to the email system, privacy concerns are avoided. We use a simpler notification construction than in Section V-D.

We include an `X-SureMailRecentIDs` header containing the smIDs of a small number (say 3) of emails sent recently, allowing the R to determine if it is missing any of those emails. Each recent smID is repeated in more than one subsequent email, which provides some tolerance to the loss of consecutive emails.

S also includes an `X-SureMailSharedID` header containing a reply-based shared secret (the smID of an earlier email it had sent and that R had replied to). Upon receiving a new email, R uses the `X-SureMailSharedID` field to check its validity and then `X-SureMailRecentIDs` to check if it is missing any email. This prevents a spammer from subverting loss detection. For an FTS, `X-SureMailSharedID` instead contains $H(S_I)$ (Section V-F.1).

Despite its ease of deployment and simplicity, in-band notifications have a disadvantage. Loss detection can only be as fast as the frequency of email exchange between S and R. Furthermore, consecutive email loss can further delay loss notification.

B. Out-of-band Notifications

In contrast, *out-of-band* (OOB) notifications are delivered via a channel that is separate from the SMTP-based email system. This decouples notification delivery from the vagaries of the email system. The detection of email loss does not have to wait for the receipt of a later email.

The OOB channel can be viewed as a storage system to which notifications are posted and from which they are retrieved using a (synchronous) put/get interface. Such a channel could be realized using, for example:

- A distributed hash table (DHT) overlaid on clients.
- A storage service such as OpenDHT [28] or commercial services such as Amazon's SQS [1].
- Dedicated notification servers for each domain to receive notifications intended for the domain's users.

We choose a design that builds on cheap, large-scale Internet-accessible storage services since these are increasingly becoming available (e.g., Amazon's SQS [1]). The simplicity of notifications and the synchronous posting of notifications (both in contrast to email) mean that the sender has a very high degree of assurance (equal to the reliability of the storage system itself) that its notification will be delivered to the recipient.

We decompose the OOB notification system into two components: a SureMail-specific service (*SM*) for the control path (e.g., handling the registration of new SureMail users) and a relatively generic storage service (*ST*) for the data path (e.g., handling the posting and retrieval of notifications). Figure 4 shows how they interact. We assume that SM and ST are operated by separate, non-colluding entities. ST can directly leverage a generic storage service. However, for full functionality, we identify a few additional capabilities and APIs we need beyond the standard put/get interface of existing storage systems. We believe they are useful for other applications as well.

The separation of SM and ST has several benefits. All of the relatively heavyweight data path workload is confined to ST, which leverages existing scalable storage services built for Internet-scale applications. The SureMail-specific SM handles the relatively lightweight control path workload, which makes it easier to build and operate. Furthermore, SM holds minimal state and is thus easy to scale with increasing load. Administrative separation between SM and ST means that neither component is in a position to learn much about the email behavior of users, even if individually they are not entirely trustworthy (Section V-A). SM knows user identities but does not see their notification workload. The converse is true for ST.

1) SureMail Service (SM): SM handles the registration of new SureMail users. This is needed to **(a)** prevent

users from accessing notifications posted for others (e.g., to monitor their volume), and **(b)** detect and block DoS attacks. Except where noted, operations on a user's behalf are performed *automatically* by their SureMail client.

As the first step, a new SureMail user's client contacts SM, giving the email address of the registering user (U). SM performs an email-based handshake (i.e., sends back an email to this address) to verify that the client "owns" the email address it is trying to register. (This is a commonly-used procedure for authenticating users, e.g., when they sign up for online services [24].) Note that this relatively heavy-weight email-based handshake is invoked just once, at the time of initial registration. If the challenge-email itself is lost, the client can retry.

To block registration attempts by automated bots, SM also returns a human-interactive proof (HIP) [6] to the new user and verifies the answer before proceeding. In this process, SM and the client establish a key, k_U, that is unique to the email address being registered and serves as the user's credentials. SM then contacts ST to create a queue to hold notifications sent to user U.

For reasons given in Section VI-B.2, k_U is constructed to be self-certifying — it is of the form $(x, MAC_z(x))$, where the key z used in the MAC is known only to ST, but not to SM or the clients. ST supplies new, valid credentials to SM, as needed, to pass on to new clients.

As explained further below, the notification queue for user U is set up to allow other users to post notifications for U but requires a key for reading these notifications. SM sets this key to k_U at the time it creates the queue for U. Possession of this key enables user U's client to directly talk to ST to change the key associated with its queue to $k_U^{'}$. Doing so prevents SM from snooping on its notifications (described in Section VI-D).

Clients renew their registrations periodically, say once a month. Renewal requires solving a new HIP. The email-based handshake and queue creation are not repeated. Renewal serves two purposes. First, it allows the system to "forget" users who have stopped using it, thereby reclaiming the corresponding resources (their storage queue). Second, it is harder for an attacker to steadily build up a large set of bogus registrations; the attacker would have to do work (solve a HIP) each time a registration expires. We believe that solving a HIP, say once a month, would not be a burden for legitimate users.

2) Storage Service (ST): ST manages the notification queues for users. It is oblivious of notifications themselves and treats them as opaque objects. It allows users to post and retrieve notifications using an interface similar to standard put/get interfaces, but with a few extensions.

First, it supports an administrative interface, which (in SureMail) only allows SM to create queues for new users. It also allows the admin to specify workload limits to block DoS attacks, as we explain below. Second, the queues provided by ST support authenticated read but allow unauthenticated writes. This is in contrast to the usual practice of authenticating writes alone or both reads and writes. ST also allows the read-key associated with a queue to be changed by presenting the current key.

We now turn to the construction and operation of ST. This service has a front end (FE) that communicates with clients and a back end (BE) that provides storage service. When it receives a client request, the FE typically invokes one or more operations on the BE. The FE only holds soft state, so it easy to scale out to keep up with load.

At user registration, SM calls on ST to create a notification queue for the user. The read-key for this queue is initially set to k_U. The user's client then contacts ST to change the key to $k_U^{'}$. This completes registration and the user is then in a position to receive notifications from others and post notifications for others. When a client posts or retrieves notifications for/to U, the FE invokes a $put()$ or $get()$ operation on the corresponding queue in the BE. The client identifies the queue using $H(U)$, so ST does not directly learn U's identity.

Despite the small size and lightweight processing of notifications, a DoS attack aimed at overwhelming the computational or storage resources of ST remains a possibility. To defend against this, SM, at system initialization time, specifies a limit on the rate of notification postings by a sender (e.g., the maximum number that can be posted in a day) and the maximum storage allowed for notifications posted by the sender (e.g., the sum of the time-to-live (TTL) values of posted notifications). It also specifies a limit on the frequency of notification retrievals by a recipient, which is enforced using a procedure analogous to the case of notification posting discussed here. ST does not enforce these limits under normal (i.e., non-attack) conditions. However, if it suspects an attack (e.g., its load is high or its storage resources are being depleted rapidly), ST enforces the per-sender limit by dropping any excess notifications. We discuss the implications of such dropping in Section VI-B.3.

To enforce per-sender limits, ST needs to identify senders in a way that is resilient to cheating. We require any client, U, that is posting or retrieving notifications to include, with its request, the original key (k_U) that it had obtained from SM at the time of registration. The HIP mechanism presents a barrier to bogus registrations on a large scale. When a client request arrives, the FE of ST first performs a quick stateless check to verify that the key is well-formed (recall from Section VI-B.1 that k_U is self-certifying), discarding the request if it isn't. Otherwise, the FE queries and updates workload/resource usage information for U in the BE and checks if any limits have been exceeded. If any have been, U's identity is pushed out to all FE nodes (as soft state, say for the rest of the day), to block further requests from U without

the need for the more expensive BE lookups.

3) DoS Defense and System Scalability: First, we consider the load placed on ST as notifications are posted and retrieved, assuming the quota checks are being performed. The posting or retrieval of a notification involves:

a) Verifying that the presented key is well-formed and, if appropriate, filtering notification postings from senders who have exceeded workload limits.

b) Retrieving the workload information for the requester, checking whether any limits have been exceeded, and storing back the updated workload information. If any limits have been exceeded, the sender is blocked and its identity is pushed out to all FE servers.

c) Storing the new notification in the case of a posting, or fetching notifications in the case of a retrieval.

Under normal operation, only step #c is performed, which involves a single $put()$ for a posting or a single $get()$ for a retrieval. When the system is under heavy load or attack, steps #a and #b are also invoked. However, we expect much of the attack traffic to be filtered in #a, either because the keys are not well-formed or because the sender has already been blocked. Note that #a is performed at the FE and does not require accessing the storage layer itself. While #b does require an additional $get()$ and $put()$, we could reduce the load significantly by performing it infrequently, say once every 10 posting/retrievals picked at random, and scaling the workload limits accordingly. Thus even when the system is under attack, the load on ST is not much more than one $put()$ or $get()$ per posting/retrieval.

A second issue is what the notification workload limits should be set to. If we set a (generous) limit of 1000 notifications per sender per day and an average TTL of 10 days, then at any point in time, ST would have up to 10,000 notifications posted by a sender. Each notification is 64 bytes in size (Section VII). This is a maximum storage of 625 KB per sender. So an ST with 1 TB of storage would support 1.72 million users. Using Amazon's S3 pricing [1] (storage rate is US $0.15 per GB per month and transfer rate of $0.20 per GB), it would cost a modest $1383 per month ($154 for storage, $1229 for transfer), or about US $0.0008 per user per month. In practice, most users may generate far fewer than 1000 notifications per day, driving costs down even further.

These calculations also suggest that it would be challenging for an attacker to consume a significant fraction of the storage resources. For instance, to consume 50% of the storage resources, the attacker would have to masquerade as over 0.8 million different senders, each of which would have had to register with the SM service and get past the HIPs.

Since the workload limits are only enforced when the system is under attack, the system is flexible in terms of the volume and the TTL of notifications under normal operation. For instance, the relatively small number of legitimate high-volume senders (such as credit card companies emailing monthly bills) can each post well over 1000 notifications per day without impacting the overall system load. Likewise, a client could choose to set the TTL for a notification to much longer than 10 days.

However, when the system is under attack and starts enforcing the limits, high-volume senders will have their attempts to post notifications temporarily blocked. To get around this, such senders would have to set up multiple registrations with SM. In any case, notifications from typical (i.e., low-volume) senders, who presumably constitute the overwhelming majority, would be unaffected even when workload limits are being enforced. Bogus notifications that are part of the attack are still not presented to the user due to our shared secret scheme.

C. Combining In-band and Out-of-Band Notifications

Since both in-band and out-of-band notifications have their advantages, our design incorporates both. In-band notifications are cheap and do not expose any information beyond the email system. Thus, we use these as the first line of defense. If an email is replied to (i.e., implicitly ACKed), there would be no reason to post an OOB notification for it. Likewise, if a NACK is received for an email (e.g., a bounceback is received or the recipient has already complained about the email being missing, say based on an in-band notification), again there isn't the need to post an OOB notification for the original email. If neither has occurred after some duration, we need to post an OOB notification, which is likely to be more reliable than an in-band notification. Analysis of the email behavior of 15 users at Microsoft suggests holding off for about 10 hours (to accommodate 75% of email replies (i.e., ACKs) for this set of users).

D. Privacy Implications of SureMail Notifications

In-band notifications do not impact privacy by design, so we focus here on OOB notifications. Our design prevents users from accessing the notification queues of other users because they do not have access to the read key for the corresponding queue (Section VI-B.2).

SM is similarly also blocked because it does not possess the read key after the user has updated it (i.e., changed it from k_U to $k_U^{'}$ per Section VI-B.2). Recall also from Section VI-B that SM and ST are assumed to be non-colluding. SM could try to cheat by changing k_U to a $k_U^{'}$ of its own choosing at the time of initial registration (or by doing so for arbitrary users who may have not even tried to register). However, doing so would prevent the legitimate U from successfully completing its registration and hence it would opt out of SureMail, leaving SM with access to an unused notification queue.

ST has access to the notifications. However, since it does not possess the reply-based shared secret between a

pair of users, it is unable to interpret them. Furthermore, notifications for U are posted to $H(U)$, so ST does not have direct access to U's identity. While, ST could try to reverse the one-way hash for users in its dictionary, this would only allow it to learn the volume of notifications posted to those users, not who posted them or what the corresponding email content is. We believe that volume information alone is not very sensitive because of notification spam or the user themselves posting fake notifications to "pad" their queue, without ST being any the wiser. Also, ST can be prevented from gleaning information from client IP addresses using anonymous communication (e.g., Crowds), if a client so desires.

VII. Implementation and Evaluation

We discuss ST & SM services to support OOB notifications and a client that also supports in-band notifications.

A. Storage Service (ST)

ST comprises a front-end that implements the extensions from Section VI-B.2, and Amazon's S3 & SQS web services as the backend. SQS [1] allows multiple values to be posted for a key and so is used for the notification queues. The simpler hash table like interface of S3 [2] is used for other persistent information (e.g., usage stats.).

1) Front-end Shim (FE): Our FE is a multi-threaded C++ program that processes requests from SM (to create queues for new users) as well as clients (to update their credentials and to post/retrieve notifications). Upon receiving a request, FE validates the credentials presented, updates usage statistics in S3 and checks against workload limits, and posts/retrieves information to/from S3 or SQS, as needed. Communication with the storage backend is secured with an access key known only to FE.

FE processing involves low overhead operations to receive/send client messages, unpack/pack the messages, and perform keyed SHA1 hashes (to check that credentials are well-formed). Ignoring the high network latency to the BE (since the FE and BE were not co-located in our setup), our prototype performed over 9000 operations per sec. of notification posts/retrievals, while saturating one core on a dual-core 3.0 GHz Pentium D with 2 GB of RAM. The FE does not hold any hard state and can easily be replicated to support higher request rates, so the scalability and reliability of the storage backend is not hampered by the additional functionality of the FE.

2) Storage Backend (BE): Based on a conversation with the designers of Amazon's S3 and SQS, we learned that both systems replicate data across storage nodes within the same data center as well as across different data centers. Upon failure of any node or data center, the data is re-replicated. The system is designed to meet 99.99% availability. However, the designers did indicate the reliability target other than to "never lose data".

Queues	44
Notifications posted	207752
Notifications retrieved	207747
Reliability	99.9976%

TABLE VI. Notification Storage Experiment Results

During email loss experiment 3 in Table I, we evaluated the suitability of SQS as the notification storage backend. For each email sent, we pushed a corresponding notification onto the SQS queue for the recipient. We did not route requests via the FE prototype during this experiment. We periodically retrieved the notifications from each queue, clearing out the queue in the process. Our findings are summarized in Table VI. The 4-nines reliability achieved is 500x better than that of email itself in the same experiment (Table I). So we conclude that the SQS storage backend available today is quite reliable for our purposes. (The 5 notifications lost had been posted to 4 different queues within a 30-minute window.)

B. SureMail Service (SM)

SM is a multi-threaded C++ program that processes SureMail client registrations. On receiving a registration request, SM returns a HIP to the client over the same TCP connection. It also emails a randomly generated password string to the email address the client is attempting to register. At a later time, when the client returns this password and the HIP answer in a registration confirmation message over a new TCP connection, SM creates a queue for the user by contacting ST and returns the well-formed user credentials (obtained from ST) to the client.

C. SureMail Client

We have implemented a standalone C++ client that interacts with both SM and ST as noted above. We have used this for testing and for benchmarking. Figure 5 shows the binary format of a post notification message. The notification itself (which is what is stored by ST) is 64 bytes long but is embedded in a 124-byte message.

We have also implemented a C#-based add-in for Microsoft Outlook 2003 to enable real use of SureMail. Our add-in uses the Outlook Object Model (OOM) [7] interface to intercept various events (e.g., email send, receipt, reply) and the Messaging API (MAPI) [25] to add x-headers. For compactness, we coalesce the various x-headers from Section VI-A into a single `X-SureMail`header that includes the message ID, recent IDs, in-reply-to ID, and shared secret. As of publication, the add-in implements the reply-based shared secret scheme and supports in-band notifications. Support for OOB notifications in this add-in is in progress.

VIII. Discussion

A. When Should the Recipient be Alerted to Loss?

A SureMail notification could arrive a few moments before the respective email. During that time, the email

Byte 0	Byte 1	Byte 2	Byte 3
Version	PktType	PktSize	TTL
CredX (4 rows = 16 bytes)			
CredY (5 rows = 20 bytes)			
H(RecptEmailAddr) (5rows)			
Timestamp (T)			
$H(smID_{M_{new}})$ (5 rows)			
$H(smSS_1)$ (5 rows)			
$MAC_{smSS_1}(T, H(smID_{M_{new}}))$ (5 rows)			

Fig. 5. Post Notification Message Format: The top half contains the credentials and the key under which the notification (bottom half) is to be stored (see Sections V-D and VII-A).

is merely delayed and not lost, and thus the receiver should not be falsely alerted to email loss. For each of the 138,944 emails sent in Experiment 1 (Section III), we calculate the end-to-end delay from when it was sent by our program to the timestamp inserted by the receiving account's email server. Almost a third of the non-lost emails have slightly negative delays, due to the lack of clock synchronization between our sender and recipient MTAs. Of the rest, the median delay is 26 seconds, mean is 276 seconds, standard deviation is 55 minutes, and maximum is 36.6 hours. Thus we believe 2 hours is an appropriate duration to wait before alerting the user.

B. Should Emails also be Delivered via SureMail?

Emails themselves should *not* be delivered via the SureMail OOB channel because doing so would fundamentally alter the nature of the channel. The rich and extensible nature of email would necessitate filtering to block malware, which is not needed with our tiny, fixed-format notifications. Emails also tend to be much larger and so would impose a far greater overhead on the OOB channel. The median email size (including attachments) across 15 user mailboxes we analyzed at Microsoft was 4 KB and the 95th percentile was 44 KB. In contrast, our notification payload is 64 bytes (Section VII).

C. Supporting Email Lists, One-Way Communication

There are cases such as mailing lists and one-way communication (e.g., email bank statements) where the normal reply-based handshake may not work. Instead, we could have a shared secret be set up, say via the usual email address validation handshake at sign-up. For a mailing list, this shared secret could be shared across all members and a common list-wide queue used for posting and retrieving notifications. In the bank case, the shared secret would be unique for each user and notifications would be posted to the individual user queues.

IX. Conclusion

Our measurement study shows that on average, 0.71% to 1.02% of email is lost silently. We have designed and prototyped SureMail. It complements the current email infrastructure and increases reliability by notifying recipients about email loss. This notification provides significant user value because the informed user can contact the identified sender for the missing information, say over phone or IM. SureMail supports in-band and out-of-band notifications with no changes to the existing email infrastructure and without PKI/PGP. It places minimal cognitive load on users. It includes mechanisms to defend against notification spam and breaches to user privacy. In our evaluation, SureMail ensured that silent email loss was detected with 99.9976% reliability.

Acknowledgements

We thank H. Balakrishnan (MIT), P. Barford (Wisconsin), C. Dovrolis (GaTech), R. Govindan (USC), S. Keshav (Waterloo), L. Qiu (UT Austin), J. Rexford (Princeton), H. Schulzrinne (Columbia), D. Veitch (Melbourne), and L. Zhang (UCLA) for helping us obtain email accounts for our experiments.

References

[1] Amazon Simple Queue Service. *http://aws.amazon.com/sqs*.
[2] Amazon Simple Storage Service. *http://aws.amazon.com/s3*.
[3] AOL Missing Email Self Help. *http://postmaster.info.aol.com/selfhelp/mbrmissing.html*.
[4] AOL Whitelist Information. *http://postmaster.info.aol.com/whitelist*.
[5] ePOST Serverless Email System. *http://www.epostmail.org/*.
[6] Human Interactive Proofs. *http://www.aladdin.cs.cmu.edu/hips/*.
[7] Outlook Object Model. *http://msdn2.microsoft.com/en-us/library/ms268893.aspx*.
[8] Pivotal Veracity. *http://www.pivotalveracity.com/*.
[9] Sender policy framework (SPF). *http://www.openspf.org/*.
[10] Silent Email Loss by EarthLink. *http://www.pbsf/cringely/pulpit/2006/pulpit_20061201_001274.html*.
[11] Spamassassin. *http://spamassassin.apache.org/*.
[12] U.C. Berkeley Enron Email Analysis Project. *http://bailando.sims.berkeley.edu/enron_email.html*.
[13] Vipul's razor. *http://razor.sourceforge.net/*.
[14] B. Adida, S. Hohenberger, and R. Rivest. Fighting phishing attacks: A lightweight trust architecture for detecting spoofed emails. In *DIMACS Wkshp on Theft in E-Commerce, April 2005*.
[15] M. Afergan and R. Beverly. The State of the Email Address. *ACM CCR*, Jan 2005.
[16] S. Agarwal, D. A. Joseph, and V. N. Padmanabhan. Addressing email loss with SureMail: Measurement, design and evaluation. In *Microsoft Research Tech. Report MSR-TR-2006-67*, May 2006.
[17] S. Agarwal, V. N. Padmanabhan, and D. A. Joseph. SureMail: Notification overlay for email reliability. In *ACM HotNets-IV Workshop*, Nov 2005.
[18] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *ACM WPES*, 2004.
[19] M. Ceglowski and J. Schachter. Loaf. *http://loaf.cantbedone.org/*.
[20] H. Ebel, L.-I. Mielsch, and S. Bornholdt. Scale-free topology of e-mail networks. *Physical Review E66*, Feb 2002.
[21] R. Fajman. An Extensible Message Format for Message Disposition Notifications. *RFC 2298, IETF*, Mar 1998.
[22] M. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. *EUROCRYPT*, 2004.
[23] M. J. Freedman, S. Garriss, M. Kaminsky, B. Karp, D. Mazieres, and H. Yu. Re: Reliable email. *USENIX/ACM NSDI, May 2006*.
[24] S. L. Garfinkel. Email-based identification and authentication: An alternative to PKI? *j-IEEE-SEC-PRIV*, Nov/Dec 2003.
[25] I. D. la Cruz and L. Thaler. *Inside MAPI*. Microsoft Press, 1996.
[26] A. Lang. Email Dependability. Bachelor of Engineering Thesis, The University of New South Wales, Australia, Nov 2004. *http://uluru.ee.unsw.edu.au/~tim/dependable_email/thesis.pdf*.
[27] A. Mislove, A. Post, C. Reis, P. Willmann, P. Druschel, D. Wallach, X. Bonnaire, P. Sens, and J. Busca. POST: A Secure, Resilient, Cooperative Messaging System. *HotOS*, May 2003.
[28] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and Its Uses. *SIGCOMM*, Aug 2005.

Wresting Control from BGP: Scalable Fine-grained Route Control

Patrick Verkaik*, Dan Pei†, Tom Scholl†, Aman Shaikh†,
Alex C. Snoeren*, and Jacobus E. van der Merwe†
*University of California, San Diego †AT&T Labs – Research

Abstract

Today's Internet users and applications are placing increased demands on Internet service providers (ISPs) to deliver fine-grained, flexible route control. To assist network operators in addressing this challenge, we present the Intelligent Route Service Control Point (IRSCP), a route control architecture that allows a network operator to flexibly control routing between the traffic ingresses and egresses within an ISP's network, without modifying the ISP's existing routers. In essence, IRSCP subsumes the control plane of an ISP's network by replacing the distributed BGP decision process of each router in the network with a more flexible, logically centralized, application-controlled route computation. IRSCP supplements the traditional BGP decision process with an *explicitly ranked decision process* that allows route control applications to provide a per-destination, per-router explicit ranking of traffic egresses. We describe our implementation of IRSCP as well as a straightforward set of correctness requirements that prevents routing anomalies. To illustrate the potential of application-controlled route selection, we use our IRSCP prototype to implement a simple form of dynamic customer-traffic load balancing, and demonstrate through emulation that our implementation is scalable.

1 Introduction

Given the best-effort communication model of the Internet, routing has historically been concerned with connectivity; that is, with finding a loop-free path between endpoints. Deviations from this default behavior normally involved policy changes at fairly slow time-scales to effect business and network management objectives [7]. Within a particular network (or autonomous system), routing was realized by a fixed, fairly simple BGP decision process that tries to ensure consistent decision making between the routers in the network, while respecting the network operator's policies.

As networked applications and traffic engineering techniques have evolved, however, they place increasingly sophisticated requirements on the routing infrastructure. For example, applications such as VoIP and online gaming can be very sensitive to the characteristics of the actual chosen data path [8, 9, 22]. A number of studies have shown that non-default Internet paths can provide improved performance characteristics [1, 2, 26], suggesting the potential benefit of making routing aware of network conditions [11]. Additionally, today's operators often wish to restrict the any-to-any connectivity model of the Internet to deal with DDoS attacks. Finally, in some cases the default BGP decision process is at odds with provider and/or customer goals and may, for example, lead to unbalanced egress links for customers that are multi-homed to a provider [29].

These demands have in common the need for route control that is (i) fine-grained, (ii) informed by external information (such as network conditions), and (iii) applied at time-scales much shorter than manual routing configuration changes (i.e., is "online") and therefore implemented by means of a *route control application*. Unfortunately, BGP does not provide adequate means for performing online, informed, and fine-grained route control. The tuning parameters BGP does provide are both arcane and indirect. Operators and developers of route control applications are forced to manipulate BGP route attributes in cumbersome, vendor-specific router configuration languages at a low level of abstraction, frequently leading to ineffective or, worse, incorrect decisions [20].

We address these requirements by presenting the design and implementation of a distributed Intelligent Route Service Control Point (IRSCP), which subsumes the routing decision process in a platform separate from the routers in a network. We previously introduced the concept of a centralized Route Control Platform (RCP) that is separate from and backwards compatible with existing routers [12]. In more recent work, we also demonstrated that route selection could be informed by "network intelligence" to enable sophisticated connectivity management applications [29]. To reflect this thinking we have changed the name of our architecture to Intelligent Route Service Control Point (IRSCP). The work presented in this paper builds on this earlier work and offers three important new contributions in the evolutionary path towards a new routing infrastructure:

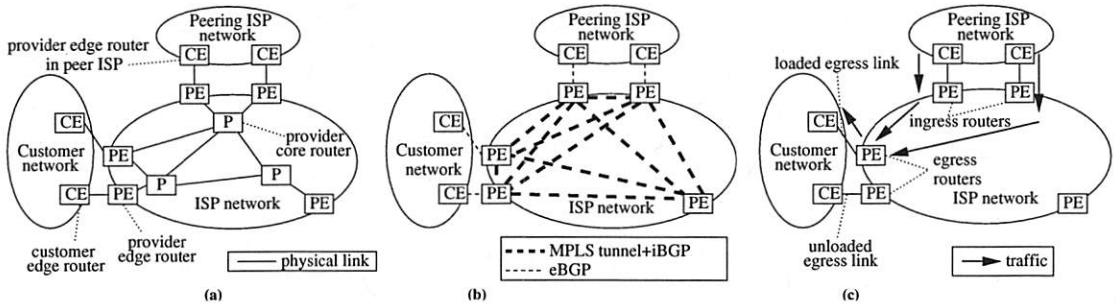


Figure 1: ISP routing infrastructure. (a) Physical connectivity. (b) BGP and MPLS. (c) Traffic ingresses and egresses.

Application-directed route selection: IRSCP eases the realization of applications that dynamically manage connectivity. Specifically, we allow an ISP's *route control application* to directly control route selection through an intuitive, vendor-independent interface. The interface is based on the abstraction of a *ranking* of egress routes for each router and destination.

Complete route control: IRSCP maintains *complete* control of the route selection function for all routers in the network. As we argue later, this can only be achieved by having IRSCP communicate directly with routers in neighboring networks via eBGP, in addition to speaking iBGP with the routers in the IRSCP-enabled network[1]. This gives IRSCP *full visibility* of all routes available in the network. Further, IRSCP is now the *sole* controller of BGP route selection, meaning that all of the network's routing policy can be handled by the route control application through IRSCP, as opposed to placing some policy configuration on the routers themselves, as is the case in an iBGP-speaking IRSCP [12].

Distributed functionality: Realizing an IRSCP that has complete control of route selection faces two important challenges. First, because routers in the IRSCP-enabled network completely rely on the IRSCP for routing decisions, the IRSCP infrastructure must be substantially more robust than an iBGP-speaking IRSCP. Second, the need for complete route control and full visibility poses significant scalability challenges. To address these concerns, we partition and distribute the IRSCP functionality across a number of servers while still ensuring consistent decision making for the platform as a whole. Although the IRSCP is physically distributed, it presents a *logically centralized* abstraction from the point of view of a route control application. I.e., while the route control application has to interact with all IRSCP servers, it can remain agnostic to where these servers reside in the network and what set of routers each server controls.

We present a modified BGP decision process, which we call the *explicitly ranked decision process*, together with a route control interface that enables route control applications to directly guide the route selection process in IRSCP. The key challenge to modifying the BGP decision process is to ensure that the resulting protocol retains (or improves) BGP's robustness, scalability, and consistency properties. We present two simple constraints on the application-provided route ranking that together ensure that IRSCP installs only safe routing configurations, even in the face of router failures or dramatic changes in IGP topology. We see this as a first step towards a "pluggable" route control architecture in which the route control application consists of several independently developed modules and uses the constraints to resolve conflicts, ensuring that only safe routing configurations are sent to IRSCP. We demonstrate the effectiveness of IRSCP's route control interface by evaluating a sample route control application (consisting of a single module) that uses IRSCP's interface to load balance customer traffic [29]. Finally, we show through experimentation that our prototype implementation is capable of managing the routing load of a large Tier-1 ISP.

2 Background and motivation

We begin by providing a brief overview of routing and forwarding in a modern MPLS-enabled ISP network. We then motivate the need for application-directed route selection.

2.1 Current operation

Figure 1(a) shows a simplified view of the physical infrastructure of an MPLS-enabled ISP backbone. The routers at the periphery of the ISP network connect to other ISPs (called peers) and customers. These routers are termed *Provider Edge (PE)* routers, and the routers that interconnect the PE routers are called *Provider Core (P)* routers. The customer routers connecting to PEs are called *Customer Edge (CE)* routers. For simplicity we also use *CE* to represent *peer* routers that connect to the ISP. BGP allows an ISP to learn about destinations reachable through its customers and peers. Typically every PE maintains BGP sessions with its attached CEs, and also with other PEs in the ISP network; the former are known as *eBGP* (external BGP) sessions and the latter as

BGP Decision Process (Section 2.1)	Explicitly Ranked DP (Section 2.2)
0. Ignore if egress router unreachable in IGP 1. Highest local preference 2. Lowest AS path length 3. Lowest origin type 4. Lowest MED (with same next-hop AS)	(*same*)
B-5. eBGP-learned over iBGP-learned B-6. Lowest IGP distance to egress router B-7. Lowest router ID of BGP speaker	R-5. Highest explicit rank R-6. Lowest egress ID

Table 1: Left: the steps of the BGP decision process. Right: the steps of our *explicitly ranked decision process*. Steps 0-4 are identical and produce the *egress set*.

iBGP (internal BGP) sessions, as shown in Figure 1(b). When a PE router receives a route through its eBGP session, it propagates the route to other PEs through iBGP sessions, allowing every PE to learn how to reach every peer or customer network. The path between traffic *ingress* and traffic *egress* routers is determined by another routing protocol known as an interior gateway protocol (IGP), such as OSPF. In an MPLS network, label-switched paths are established between all PEs (see Figure 1(b)), obviating the need to run BGP on P routers.

A PE usually receives more than one egress route for a given destination and must run a route selection algorithm called the *BGP decision process* to select the best route to use for data forwarding. The BGP decision process (shown in the first column of Table 1) consists of a series of steps. Starting with the set of routes available to the PE, each step compares a set of routes and passes the most preferred routes to the next step while discarding the remaining routes. Steps 1–4 compare routes in terms of *BGP attributes* attached to the routes, while steps 0 and B-6 consider the IGP information associated with the egress PE of the route. We call the set of routes remaining after Steps 0–4 the *egress set*. Steps B-5 and B-6 are responsible for what is called *hot-potato routing*, i.e., forwarding traffic to the nearest (in terms of IGP distance) egress PE in the egress set. Step B-7 is a tie-breaker that ensures that the PE always ends up with a single best route.

2.2 Application-directed route selection

We use Figure 1 to highlight a specific problem introduced by BGP's hot-potato routing thereby motivating the need to enhance route selection with fine-grained application-directed route control.[2] We assume that the customer shown in Figure 1 has multihomed to the provider network for the purpose of improving redundancy, and so the same destinations are reachable via both links. All PEs in the provider network therefore have two possible routes to reach the customer network. Assume further that most of the traffic destined to the customer network enters the provider network from the peering ISP network. Assuming unit IGP costs for each internal provider link in Figure 1(a), the two ingress PEs at the top of the figure prefer the route via the top egress PE connected to the customer network (Figure 1(c)). This leads to an imbalance in the load on the two egress links, with the top egress link carrying all (or most) of the traffic, which in turn may result in congestion.

In practice the customer or ISP may prefer the ISP to load-balance traffic on the two egress links while still considering the IGP path cost between ingress and egress. In IRSCP this goal is achieved by basing the decision process on the input from a load-balancing *route control application* run by the ISP, which takes into account IGP path cost, as well as "offered" ingress load and capacity of egress links. The route control application directly controls route selection for each PE and, in this example, directs the two ingress PEs to each send traffic to a different egress PE. We emphasize that this routing solution cannot be achieved in BGP without manipulating BGP attributes on multiple PEs through vendor-specific configuration languages. In contrast, IRSCP exports a simple, intuitive interface that allows a route control application to supply a *ranking* of egress links that is evaluated during execution of a modified decision process, the *explicitly ranked decision process*. The interface effectively isolates complexity and intelligence in the route control application, while IRSCP itself remains simple. As shown in Table 1, the explicitly ranked decision process replaces the hot-potato steps (B-5 and B-6).

3 Scalable route control

In this section we describe the design of a distributed Intelligent Route Service Control Point (IRSCP). We begin by presenting our modified route selection process that gives route control applications the ability to direct traffic entering the network at a given ingress PE to an arbitrary egress link. The second section presents the architecture of IRSCP, discussing issues of scalability and fault-tolerance. We formulate a consistency requirement for the explicitly ranked decision process that prevents forwarding anomalies and show that enforcing simple constraints on the application input is sufficient to satisfy the requirement.

3.1 Explicitly ranked decision process

IRSCP implements two types of decision process: the normal BGP decision process and the *explicitly ranked decision process*. Both perform route selection on behalf of individual PEs and so are defined on a per-PE basis. The BGP decision process is used for the subset of destinations, *unranked prefixes*, for which the customer, ISP or route control application has determined that conven-

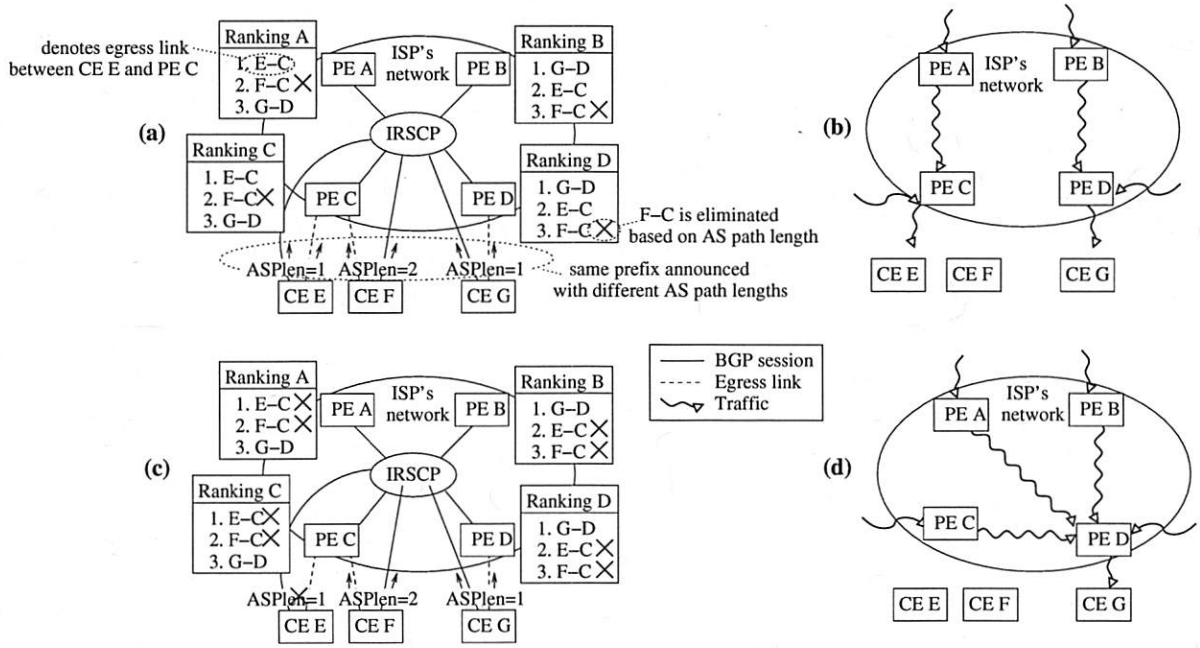


Figure 2: Example of ranking and its effect on forwarding. The application (not shown) has provided IRSCP with the explicit rankings indicated. Each ranking is a list of egress IDs. CEs E, F and G announce routes for the same prefix (with different AS path lengths) through their eBGP sessions with IRSCP. (a) Initial scenario. (b) Forwarding behavior for (a). (c) CE E withdraws its routes. (d) Resulting forwarding behavior.

tional hot-potato routing can be used. For the remaining *ranked prefixes*, the route control application creates a preference ranking of egress routes for each ingress router, the selection of which is realized in IRSCP by the explicitly ranked decision process.

In our architecture the route control application tells IRSCP which routers should use which egress routes based on routing information, traffic load measurements, etc. As a general principle IRSCP follows the directives of the application, except when doing so severely impairs connectivity. An instance of this principle is that we let the application specify a *ranking* of egress links, i.e., egress links ranked by preference, rather than a fixed assignment of egress links to routers. (Each egress route corresponds to only one egress link, and we therefore use the terms egress link and egress route interchangeably.) Using a ranking accommodates unavailability of egress routes. For example, if the top-ranked egress route is unavailable, the next-ranked egress route may be selected. The application specifies the ranking on a per-destination, per-router basis.

We construct our decision process for ranked prefixes (Table 1) by adopting Steps 0–4 of the BGP decision process and then apply the explicit ranking instead of performing hot-potato routing, followed by a tie-breaker in Step R-6. This ensures that the explicitly ranked decision process respects BGP attributes such as AS path length (Step 2) and that it takes reachability of egress routers into account. In principle, the explicit ranking can be applied at any point in the decision process, e.g., it may override earlier steps of the decision process. However, we leave exploring the extent to which we may safely override or replace BGP attributes to future work. As we explain in Section 3.2.4, we specifically do not include a step based on IGP distance (Step B-6).

We explore an example of the explicitly ranked decision process by considering the scenarios shown in Figures 2(a) and (b). In this example a single IRSCP server runs the decision process for every PE in the ISP's network. We examine the execution of the decision process for PE A in Figure 2(a). First the IRSCP server receives all routes for the given prefix: $E-C$, $F-C$ and $G-D$. (We refer to each route using its *egress ID*, the pair of (CE,PE) routers incident on the egress link for the route.) Next, the explicitly ranked decision process for PE A executes Steps 0–4 and in Step 2 eliminates egress route $F-C$ based on the longer AS path length. (We assume that the routes otherwise have identical BGP attributes.) The result is the egress set $\{E-C, G-D\}$. In Step R-5 the decision process applies the explicit ranking for PE A to the egress set. Since the top-ranked egress link $E-C$ is present in the egress set, the decision process selects this route for PE A. Similarly, the decision process selects route $E-C$ for PE C, and route $G-D$ for PEs B and D, resulting in the forwarding behavior shown in Figure 2(b). An important observation is that Steps 0-4 are identical for all PEs. Therefore the decision process for any PE computes the same egress set.

3.1.1 Outdated rankings

Ideally, the input from the application to IRSCP continuously reflects the current state of the network. In practice, however, IRSCP, being an active participant in BGP, is in a better position to respond instantly to changes in routing state such as BGP route attributes (using Steps 0–4 of the decision process), IGP distances (discussed in Section 3.2.4), and the availability of BGP routes (discussed below). IRSCP must therefore adapt the rankings from the application (based on possibly outdated routing information) to current routing information, as follows.

Between the time that the application sends the rankings to IRSCP and the time that the explicitly ranked decision process runs, new egress routes may be announced and old routes may be withdrawn. Until the application updates its rankings, IRSCP must accommodate discrepancies between the available routes assumed when the application creates the rankings and the actual available routes. An instance of a withdrawn egress route is illustrated in Figures 2(c) and (d), in which CE E withdraws egress route $E - C$, and the egress set changes to $\{G - D\}$. As a result, the decision process changes its selection for PEs A and C to $G - D$ and all traffic egresses through PE D (Figure 2(d)). In other words, a ranking specifies not only the desired routing for the PE in the absence of failure, but also the desired fail-over behavior that the PE should adopt.

Conversely, if new egress routes are advertised, IRSCP appends them to the end of the explicit ranking (in order of egress ID) until the application is able to provide a revised ranking (Steps R-5 and R-6). Alternatively, the application may *prevent* IRSCP from appending routes in this manner. For example, the application may wish to restrict the set of egress routes of a particular customer to a fixed set, thereby preventing some forms of prefix hijacking. We define a "virtual" *black-hole egress route*, which is part of every egress set and (conceptually) sinks traffic directed to it. We also define a corresponding *black-hole egress ID*, which an application can include as part of a PE's ranking. If the explicitly ranked decision process for a PE selects the black-hole egress route, the IRSCP server does not send a route to the PE (or its attached CEs), thus making the destination unavailable through that PE.

3.1.2 Other IRSCP applications

By opening up the decision process to external input, IRSCP enables a class of applications in which routing is controlled based on information external to BGP. An example of an application that uses external information (or in this case analysis) is presented in [18]. The authors propose a pragmatic approach to BGP security by which suspicious routes are quarantined for a certain period of time before being considered for selection. In this case knowledge about the historical availability of routes (even if the routes were not selected) is important to determine whether a route is potentially hijacked. Further, IRSCP provides a mechanism (the black-hole egress route) by which routers can be prevented from selecting suspicious routes until deemed safe by the application.

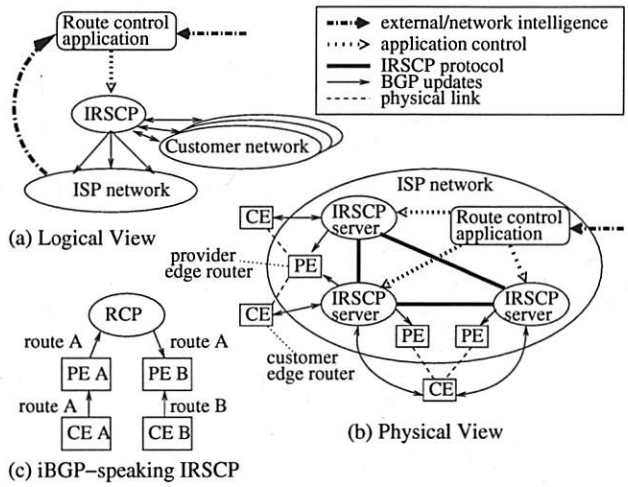


Figure 3: Overview of the IRSCP architecture.

Another example in this category is the load-balancing application described above, which makes use of network conditions inside the IRSCP-enabled network to inform route selection. However, it is also possible to inform route selection with network conditions external to the IRSCP-enabled network. For example, Duffield *et al.* [11] explore the possibility of using measured path conditions to select between various alternate paths leading to the same destination. This allows the network to route traffic that is sensitive to adverse network conditions along paths more favorable than those that default BGP would be capable of finding (to the extent permitted by the policies encoded in BGP attributes).

The complete route visibility afforded by IRSCP also simplifies a number of route monitoring applications. For example, auditing applications that ensure that peers abide by peering agreements require complete visibility of the routes being advertised to a network [23]. Similarly, "what-if" network analysis [13] becomes much simpler if the application is aware of all the routes that were available. IRSCP's ability to use external input to inform route selection, however, is its key advantage.

3.2 IRSCP architecture

Figure 3 contrasts logical and physical views of the IRSCP architecture. In Figure 3(a) an application uses "external information" to inform route selection in IRSCP, which in turn communicates selected routes to the routers in the ISP network and in neighboring ISP

networks. Figure 3(b) shows a simplified physical realization of the IRSCP architecture, consisting of the route control application and a distributed set of IRSCP servers that collectively perform route selection. The function of IRSCP is to compute a routing solution in which egress routes received from CE routers are assigned to PE routers, so that a PE router will forward traffic it receives along the route. IRSCP performs this function by communicating with PEs and CEs using standard BGP, as a result of which customers and peer networks need not be aware that their CEs peer with IRSCP rather than BGP routers. Specifically, IRSCP receives BGP routes from CE routers over eBGP, executes a per-PE decision process to determine what routes each PE should use, and sends the resulting BGP routes to the PEs over iBGP. IRSCP also sends an update to each CE attached to a PE (again over eBGP) that corresponds to the routing decision that was made for the PE.

We use Figure 3(c) to emphasize the importance of implementing an *eBGP-speaking* IRSCP in order to establish full route control. The figure shows an IRSCP that only speaks iBGP (similar to RCP [6]). For clarity we refer to such an iBGP-speaking IRSCP as RCP. RCP exchanges routes with PEs using iBGP and never communicates with CEs directly. Suppose that RCP sends an iBGP update to PE B containing route A. To implement full route control, the update must override any routes that PE B receives from other CEs (CE B). However, this implies that PE B never sends alternative routes (route B) to RCP unless route A fails (or changes its attributes). RCP is thus deprived from using any but the first route it learns. We conclude that an iBGP-speaking IRSCP restricts the ability to apply full route control.

3.2.1 Distribution

Though logically centralized from a route control application viewpoint, IRSCP is implemented as a distributed system—consisting of multiple IRSCP servers—to address fault-tolerance and scalability requirements. If we designed IRSCP as a single centralized server, failure or partitioning away of that server would leave every PE in the network steerless and unable to forward traffic correctly, since in BGP a session failure implicitly causes BGP routes announced through the session to be withdrawn. In IRSCP we can tolerate the failure of an IRSCP server by letting routers peer with multiple IRSCP servers. Furthermore, a centralized IRSCP faces a number of (overlapping) scalability challenges. First, a large Tier-1 ISP's routers collectively maintain many thousands of BGP sessions with routers in neighboring networks, something that no current BGP implementation is able to support by itself. Second, IRSCP makes routing decisions for the hundreds of PEs within the ISP. Third, IRSCP must store the combined BGP routes received from CEs and originated by the network, and it must process updates to these routes.

As shown in Figure 3, we choose to partition the IRSCP workload by letting each IRSCP server peer with a subset of PEs and CEs, thereby addressing the first two scalability concerns. Our architecture still requires each IRSCP server to store all BGP routes and process the corresponding updates; however, we show in Section 5 that this aspect of scalability does not pose a severe problem.

In performing the per-PE decision process, an IRSCP server needs to take into account the position of that PE in the IGP topology. To maintain consistency of IGP routing state, each IRSCP server runs an IGP viewer and has the same global view of the IGP topology [6]. Due to the partitioning of work in our distributed architecture, each IRSCP server needs to perform shortest-path calculations only for the set of PEs for which it makes BGP routing decisions rather than for the network as a whole. The IRSCP servers further ensure consistency by exchanging *all* BGP updates among each other. Comparing this solution with BGP route reflection [4], the most important difference is that a route reflector selects a *single route* as best route for each destination (using the "normal" BGP decision process) and only makes that route available to other routers and route reflectors. As a result different routers and route reflectors may observe a different set of available routes, which in turn has led to non-deterministic or divergent behavior in BGP, e.g., "MED oscillation" [15, 21].[3] Basu *et al.* [3] show that exchanging all routes selected by Steps 0-4 of the decision process is sufficient to prevent non-determinism and divergence in iBGP; IRSCP exchanges a superset.

Thus we propose a simple and elegant solution to distribution: a full mesh in which all IRSCP servers distribute all routes. Architecturally, this solution is similar to an iBGP infrastructure in which BGP routers are fully meshed, an architecture that was eventually replaced by route reflectors due to scalability problems. However, there are two differences between the two architectures. First, the number of IRSCP servers is small compared with the number of routers. Second, being based on commodity technology rather than router technology, we expect IRSCP servers to keep better pace with the technology curve than routers have [17].

3.2.2 IRSCP protocol and RIB

The *IRSCP protocol* (Figure 3(b)) is responsible for ensuring route exchange among IRSCP servers and is implemented as a simple extension to BGP. Each pair of IRSCP servers maintains a TCP-based *IRSCP session* through which they exchange incremental updates in the form of advertisements and withdrawals of routes. At

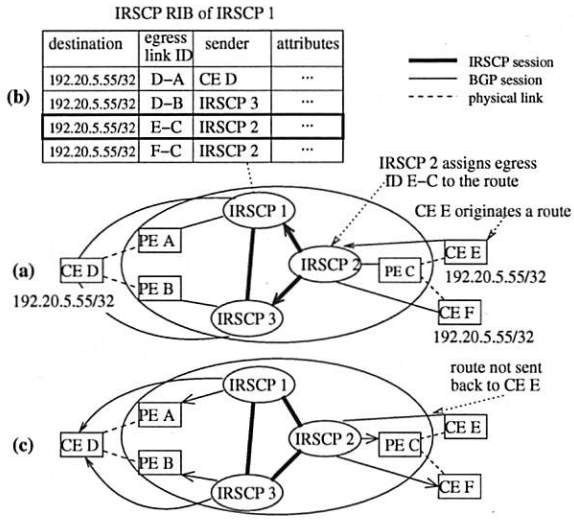


Figure 4: IRSCP route propagation example.

session startup the IRSCP servers exchange advertisements corresponding to all known BGP-learned routes, similar to "normal" BGP sessions. When an IRSCP session goes down, all routes exchanged previously on the session are implicitly withdrawn. When an IRSCP server learns a route from a CE router, it sends the route to all other IRSCP servers through the IRSCP sessions.

Figure 4 shows an example where the same destination prefix (192.20.5.55/32) is advertised by three different CEs connected to three different PEs. As shown in Figure 4(a), IRSCP 2 learns two routes to destination 192.20.5.55/32 through BGP, and it must send both instances to IRSCP 1 and 3. To distinguish several routes for the same destination, the IRSCP protocol includes in each update an identifier for the egress link to which the route corresponds. The *egress link identifier* (*egress ID* for short) is a (CE,PE) pair of the routers incident on the egress link. For example, the routes learned through IRSCP 2 have egress IDs $E - C$ and $F - C$.

When an IRSCP server receives routes from BGP routers and from other IRSCP servers, it stores the routes in a routing information base (RIB), so that the routes are available to the decision process and for further propagation to routers and IRSCP servers. For example, the IRSCP RIB of IRSCP 1 shown in Figure 4(b) contains four entries for prefix 192.20.5.55/32. Each entry has fields for the destination prefix, the egress ID, the neighbor of the IRSCP server from which the route was received, and BGP attributes that belong to the route. In the example, CE D has sent routes to IRSCP 1 and 3, resulting in the first two entries. The last two entries correspond to routes sent to IRSCP 2 by CEs E and F.

Based on the RIB, an IRSCP server executes a decision process and sends a single route per destination (Figure 4(c)) to each attached BGP router. Since the

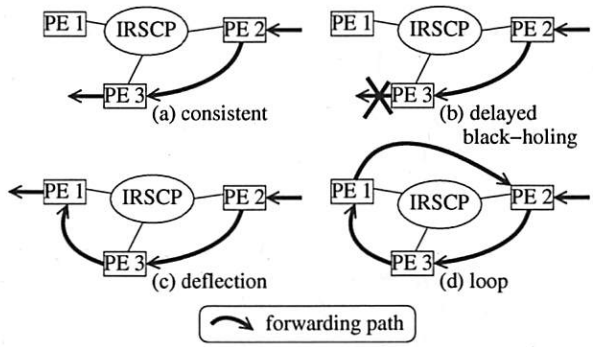


Figure 5: Forwarding anomalies. In all cases the decision process for PE 2 has selected an egress route through PE 3 as the best route for some destination. (a) The decision process for PE 3 has selected a local egress route as best route, and therefore is consistent. (b) The decision process for PE 3 has not selected any route for this destination, thus traffic is black-holed. (c) The decision process for PE 3 has selected PE 1 as best egress route, resulting in a deflection. (d) The forwarding loop is a result of multiple deflections.

IRSCP server only advertises one route per destination through each BGP session, the egress link ID is not needed (nor recognized by BGP) and so is stripped before sending the route.

3.2.3 Ensuring consistency

The concept of application-provided explicit rankings permits a route control application a great deal of flexibility. However, it also introduces the possibility of IRSCP executing the decision process in an inconsistent manner for different PEs, which can lead to forwarding anomalies. Figure 5 depicts the forwarding anomalies that may result: delayed black-holing, deflection, and forwarding loops. A packet is said to be deflected if a router on its forwarding path chooses to forward to a different egress router than the egress router previously selected by a router upstream on the forwarding path [16]. In an MPLS network deflections only occur at egress routers. We wish to prevent deflection for two reasons. First, given the existence of shortest-path MPLS tunnels between two PEs, forwarding through an intermediate BGP router is suboptimal (Figure 5(c)). Second, uncontrolled deflection can lead to a forwarding loop (Figure 5(d)). Similarly we wish to avoid delayed black-holing as in Figure 5(b) since it is wasteful to carry traffic through the network only to have it dropped. If the intent is for the traffic to be dropped, it should be dropped on ingress (i.e., at PE 2).

Ultimately, the *correctness* of the rankings is specific to the application. However we consider *consistency* to be a minimum standard of correctness for any route control application and therefore define a set of per-

destination constraints on any set of rankings provided by an application. Enforcing these constraints (by an application or by a resolver in a pluggable route control application) ensures that the explicitly ranked decision process is *deflection-free*, i.e., free of deflections and delayed black-holing.

We define the operator $<_r$ as: $e_1 <_r e_2$ iff in the explicit ranking for router r egress link e_1 is ranked above egress link e_2. For example, for PE A in Figure 2(a), we have $E-C <_A F-C$ and $F-C <_A G-D$.

Definition: *Ranking-Consistent-1:* The set of egress routes appearing in each router's explicit ranking is identical.

Definition: *Ranking-Consistent-2:* For each router r and all egress links e_1, e_2: if $e_1 <_r e_2$ then $e_1 <_{pe(e_1)} e_2$, where $pe(e)$ is the PE incident on e.

The rankings shown in Figure 2(a) clearly satisfy Ranking-Consistent-1: all rankings contain the same egress links. They also satisfy Ranking-Consistent-2. For example, checking the ranking for PE B we see that (1) $G-D <_B E-C$ and $G-D <_D E-C$, (2) $G-D <_B F-C$ and $G-D <_D F-C$, (3) $E-C <_B F-C$ and $E-C <_C F-C$.

If the explicit rankings given to an explicitly ranked decision process satisfy Ranking-Consistent-1 and Ranking-Consistent-2 then the explicitly ranked decision process is deflection-free. We omit the proof due to space constraints; it can be found in our companion tech report [30]. In [30] we also show that the BGP decision process in IRSCP is deflection-free.

Essentially, the ranking abstraction is able to describe a preferred egress link for each PE and per-PE fail-over behavior such that traffic does not get deflected. It is powerful enough to express any consistent assignment of egress routes to routers. However, the constraints do not permit failing over from one arbitrary consistent assignment to another. For example a given set of rankings that ranks egress link e_1 highest for PE A cannot fail over in such a way that egress link e_2 is assigned to PE A, unless e_1 fails.

3.2.4 IGP reachability

We assume that the application has taken IGP distances into account when it creates the ranking. Although the IRSCP decision process could conceivably re-rank egress links in response to IGP distances, it generally does not do so for several reasons. First, for applications such as load balancing customer traffic, strict adherence to a shortest-path policy appears to be of secondary importance. Indeed, tracking IGP distance changes can have adverse effects, such as causing large volumes of traffic to shift inadvertently [28]. The explicit ranking provided by an application introduces a degree of stability, effectively "pinning" routes. If it is necessary to respond to IGP changes, we require the application to do so by providing an updated ranking. Teixeira *et al.* [27] suggest that in a large ISP with sufficient path diversity in its IGP topology the latency of MPLS tunnels is not greatly affected by IGP changes. For these cases, route pinning does not sacrifice much performance in terms of latency.

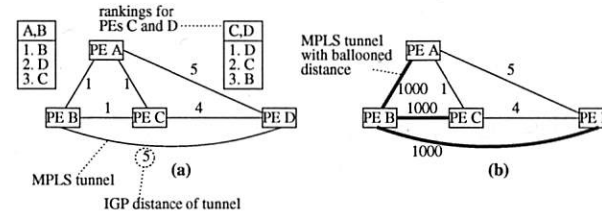


Figure 6: Example of IGP ballooning. All routers are connected by MPLS tunnels whose distance metric is computed by IGP. (a) shows a topology based on which the application has computed a set of rankings. In (b) the distance of various tunnels has increased excessively.

However, we do wish to handle the case in which IGP distances "balloon" excessively, effectively making some egress routes unusable. For example, this can occur when physical connectivity is disrupted and IGP diverts traffic around the disruption. Another example is router maintenance: typically the maintenance procedure involves setting the IGP distance between the router and the rest of the network to a very high value in order to gracefully move the traffic away from the router before it is brought down.

The network shown in Figure 6 has three egress routes for some given destination: through PEs B, C and D. The application has assigned the egress route through PE B to PEs A and B and the egress route through D to PEs C and D. In Figure 6(b) several IGP distances have ballooned, making PE A's preferred egress route through B virtually unusable for PE A, although A's rankings have not yet been updated.

We define an *Emergency Exit* procedure for cases such as this, which is as follows. If an IRSCP server finds that the IGP distance from a PE to the PE's preferred egress route balloons, the IRSCP server ignores the rankings for that PE and destination and reverts to hot-potato routing (i.e., selects the nearest egress router, possibly the PE itself).[4] In the example, PE A overrides its ranking and chooses PE C. PE C's most preferred egress route (through D) has not ballooned and therefore PE C deflects to PE D, at which point the traffic egresses.

As this example shows, ignoring the rankings may lead to a deflection. We consider this acceptable, since (a) in a well-engineered network excessive ballooning should be the exception and (b) at most one deflection

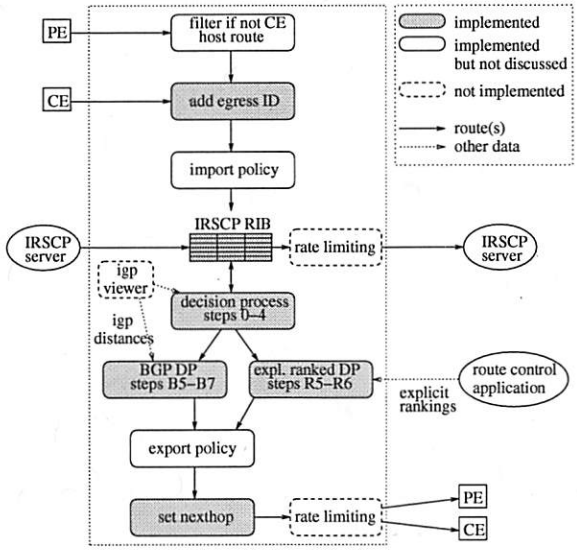


Figure 7: Control flow through an IRSCP server.

(and no delayed blackholing) can occur, and therefore no forwarding loop can occur, as we prove in [30].

4 Implementation

A significant part of the IRSCP server functionality is identical to that of a BGP router. We therefore based our prototype implementation on the version 3.9 code base of `openbgpd`.

Figure 7 summarizes the control flow among the various components that our implementation uses to receive, store, process and send routes. Note that while we have implemented an IGP viewer for a previous centralized control platform [6], we have not yet ported it to our IRSCP prototype. In addition, BGP defines a rate limiting mechanism ("MinRouteAdvertisementIntervalTimer" [24]) that we have yet to implement for BGP sessions and adapt for IRSCP sessions. A discussion of the implementation of "regular" import and export policies appears in [30].

4.1 Explicit application ranking

As shown in Figure 7, route control applications provide rankings for the IRSCP's explicitly ranked decision process. In our prototype they do so through the IRSCP configuration file, which contains a number of `rank` statements, one for each set of prefixes that are ranked identically. The following is an example of a `rank` statement for two prefixes.

```
rank {
 prefix    { 3.0.0.0/8, 4.0.0.0/8 }
 pe        { 1.2.3.4, 5.6.7.8 }
 egresses { 11.12.13.14:15.16.17.18,
            19.20.21.22:23.24.25.26 }
 pe        { 101.102.103.104 }
 egresses { 19.20.21.22:23.24.25.26,blackhole }
}
```

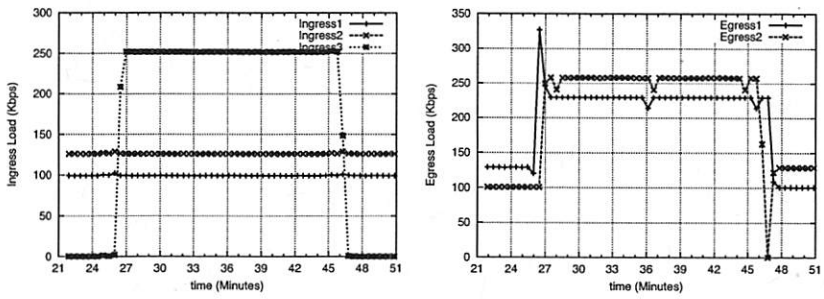
The `pe` statements of a `rank` statement must contain every PE attached to the IRSCP server. Recall that in IRSCP each route carries an egress ID, specifying what egress link traffic for the destination of the route is to use when it exits the network (assuming the route is selected as best route). The `egresses` statement is the ranking of egress IDs to be used for the PEs in the preceding `pe` statement. Each egress ID consists of the IP addresses of the CE and PE incident on the egress link. In addition, the `blackhole` egress may be specified. Application rankings are stored in a per-PE Red-Black Tree of destination prefixes, where each prefix points to the ranked list of egress IDs for that PE and prefix. When a new ranking set arrives, the IRSCP server updates its ranking tree and reruns the decision process for affected destinations.

4.2 Decision process and IRSCP RIB

The data structures used to implement the RIB (*IRSCP RIB* in Figure 7) are adapted from `openbgpd`'s implementation of the BGP RIB. `openbgpd` provides various indexes into the BGP RIB, the most important of which is a Red-Black Tree of destination prefixes, where each entry points to a list of routes (one route for each neighbor). To simplify the implementation we maintain this structure, despite the fact that the number of routes per prefix in IRSCP increases to one route per egress link. Our performance evaluation in Section 5 shows that the resulting increase in search time for a route in the RIB is not a great concern.

`openbgpd` maintains the per-destination list of routes in order of preference, based on pairwise comparison of the BGP attributes of routes according to the BGP decision process steps shown in Table 1. Thus each route update is linear in the number of routes for the destination, and the decision process itself amounts to no more than checking whether the egress router for the route at the head of the list is reachable (Step 0).

However, there are two problems with this algorithm for IRSCP. First, pairwise comparison of the MED value (Step 4) produces results that may be incorrect and, furthermore, dependent on the order in which the routes were inserted into the RIB, which in turn leads to potential inconsistency among the RIBs in different IRSCP servers. The underlying problem is that MED, being comparable only between routes from the same neighboring network, does not follow the "rule of independent ranking" [15].



(a) Ingress load

(b) Egress load

Figure 8: Functional evaluation.

The second problem is that IRSCP defines a per-PE decision process, resulting in a different order of routes for different PEs. However, we note that Steps 0–4 of the (BGP or explicitly ranked) decision process (which compute the egress set) are independent of the PE, hence we compute these steps once for all PEs (*decision process steps 0–4* in Figure 7), storing the result in the RIB. Our implementation orders routes in the RIB using pairwise comparison based on Steps 0–3. Next, of the routes ranked highest so far, it examines each set of routes received from the same neighboring network and sets a flag on those that have highest MED value in such a set, thereby computing the egress set (Step 4). The algorithm used for Step 4 is similar to that used by the Quagga open source BGP stack (`www.quagga.net`) and runs in $O(n^2)$ time for n routes available for a prefix.

Next, the PE-specific steps of the decision process are executed using a pairwise comparison of routes in the egress set. In the case of the BGP decision process (*BGP DP steps B5-B7* in Figure 7), we break the tie in Step B-7 based on the egress ID of the route rather than the router ID (of the BGP router or IRSCP server that sent the route), since a neighboring IRSCP server may have sent multiple routes for the destination. In the case of the explicitly ranked decision process (*expl. ranked DP steps R5–R6* in Figure 7), the IRSCP server adds a blackhole route (with lowest possible egress ID) to the egress set and then retrieves the list of ranked egress IDs for the PE and destination (Section 4.1). When comparing the egress IDs of a pair of routes in the egress set, our implementation simply walks down the list of egress IDs until it finds the higher ranked of the two egress IDs. This runs in time linear in the number of routes times the size of the ranking for a prefix and PE, or, if all n prefix's routes appear in its ranking, in $O(n^2)$ time.

Following (re-)execution of the decision process for a given PE, the IRSCP server distributes its decision to the PE and to all CEs attached to the PE. Note that the IRSCP server does not need to run a separate decision process for a CE: as in BGP, the route sent to the CE is the same as is selected for the associated PE.[5] To prevent unnecessary updates from being sent to neighbors, BGP implementations typically remember the last route selected for each destination as the *active route*. Our IRSCP server implementation uses the same technique; however, instead of storing a single active route, it stores an active route for each attached PE.

To evaluate scaling of the decision process in terms of the number of routing policies, we should consider not only the time needed to evaluate the ranking of a prefix (linear time, see above), but also the time needed to look up the ranking. Retrieving a ranking runs in time logarithmic in the number of ranked prefixes and (in our implementation) linear in the number of PEs per IRSCP server. This is similar to the time needed by a BGP implementation to look up a prefix in the routing table and retrieving peer state before sending updates to a peer.

4.3 Egress management

Based on the egress ID specified in each IRSCP route, IRSCP has to ensure that (a) BGP routers are "instructed" to forward the traffic towards and through the indicated egress link, and (b) the route is only used if the egress link is available. To implement (a) we use the *nexthop* BGP attribute of a route. This attribute tells a router to which *nexthop router* it must forward traffic when it uses the route. The IRSCP server sets the nexthop attribute (*set nexthop* in Figure 7) when it sends a route to a PE or CE in such a way that traffic passes through the egress link and uses the egress ID to determine what that nexthop should be. For example in Figure 2(a) and (b), IRSCP determines from egress ID $E - C$ that PE A must use PE C as nexthop, and PE C must use CE E as nexthop. In addition, a CE attached to PE A (not shown) is sent PE A as nexthop. The latter is not determined based on egress ID. Rather, as we discuss next, an IRSCP server associates each CE with a PE, and it is this PE that IRSCP sets as nexthop when sending to the CE.

When an IRSCP server is configured to form an eBGP session with a CE, part of the configuration identifies a PE with which to associate the CE: the PE to which the CE is physically attached through an egress link (e.g., see Figure 3(b)). Apart from setting the nexthop router

for routes sent to the CE (as discussed above), the IRSCP server uses the identity of the PE and CE to set the egress ID (*add egress ID* in Figure 7) on routes received from the CE.

5 Evaluation

In this section we first present a functional evaluation, demonstrating that an example load-balancing application is able to effect fine-grained route control using the ranking abstraction and our prototype IRSCP implementation. We then evaluate the scalability and performance of our prototype in processing BGP updates.

5.1 Functional evaluation

We verified the functionality of IRSCP with a testbed consisting of an IRSCP server, a load-balancing route control application, three ingress PEs, (Ingress1, Ingress2, and Ingress3), two egress PEs, (Egress1 and Egress2), a core router, and a number of hosts arranged into a "source" network and a "destination" network. The core router uses point-to-point links to connect to the PEs and we use GRE tunnels as the tunneling technology between the PEs. We assume a simple scenario in which the egress PEs attach to a customer who prefers its incoming traffic to be balanced on the egress links.

The load-balancing application performs SNMP GET queries on each PE to collect offered and egress loads. Based on the offered load, the application computes a ranking set that balances the load on the egress links. If the ranking set changes, the application generates an updated configuration and issues a configuration reload command. The hosts in the source network send constant rate UDP packets to the hosts in the destination network. Figure 8(a) shows the offered load at the ingress PEs as measured by the application server.

The load at the egress PEs is shown in Figure 8(b). Before $t = 26$, the load at these two egresses is "balanced" as a result of the initial ranking set at the IRSCP (i.e., Ingress1 prefers Egress2, and Ingress2 prefers Egress1). At $t = 26$, the offered load at Ingress3 increases, and the application takes up to 30 seconds (the SNMP polling interval) to detect this change. It then realizes that the best way to balance the load is to send the load from Egress1 and Egress2 to one egress point, and the load from Egress3 to another. Therefore, it generates a new ranking set and sends it to the IRSCP server, which updates the ingress PEs. As a result, the load from Ingress3 goes to Egress2, and the load from Ingress1 and Ingress2 goes to Egress1. Similarly, at $t = 47$ the application generates a new ranking set and shifts Ingress1 and Ingress2's loads to different egresses.

5.2 Performance measurement testbed

Rather than emulating an entire ISP infrastructure, we run our tests on a single IRSCP server, loading it as though it were part of a hypothetical Tier-1 ISP network. Note that there is limited value in emulating an IRSCP-based infrastructure for the purpose of performance evaluation. Due to the fact that IRSCP servers exchange all routes, routing in an IRSCP infrastructure does not oscillate like it does in a BGP-based infrastructure [3]. Therefore convergence time of an update is governed solely by communication latency and the processing latency of an update in an IRSCP server or a BGP router.

Figure 10(a) shows the Tier-1 ISP network modeled. Each of the n_{pop} POPs (Point-of-Presence, e.g., a city) contains one IRSCP server and all IRSCP servers are fully meshed. Assuming IRSCP as a whole processes a combined BGP update[6] rate $rate_{all}$ from all CEs, then on average an IRSCP server sends roughly $rate_{all}/n_{pop}$ to each IRSCP server, namely the share of updates that it receives from CEs in its own POP. In other words, each IRSCP server sends and receives at a rate of about $\frac{n_{pop}-1}{n_{pop}} \cdot rate_{all} \approx rate_{all}$ on its combined IRSCP sessions (Figure 10(b)). The update rate that an IRSCP server sends to a PE or CE is governed by the output of best route selection based on the (BGP or explicitly ranked) decision process. We make the simplifying assumption that this rate $rate_{best}$ is the same for all PEs and CEs. From measurements taken in a Tier-1 ISP network we derive highly conservative ball park estimates for $rate_{all}$ and $rate_{best}$ [30]. These estimates are based on the busiest day of the past year in May 2006 (in terms of total number of updates received by a route collector in the ISP's network on a day). We divide the day into 15-minute intervals and use the average of each interval to derive the estimates for that interval. From the busiest 15-minute interval we deduce an estimate for $rate_{all}$ and $rate_{best}$ of about 4900 and 190 updates/s, respectively. The 95th percentile interval gives a significantly lower estimate of about 600 and 24 updates/s, respectively.

Figure 10(c) shows our experimental setup, consisting of an update generator, the IRSCP server under test and an update receiver. The IRSCP server and update receiver are 3.6-GHz Xeon platforms configured with 4 GB of memory and a 1-Gb Intel PRO/1000MT Ethernet card. The update generator contains a 993-MHz Intel Pentium III processor, 2 GB of memory, and a 100-Mb 3Com Ethernet card. All three machines run OpenBSD 3.8.[7] The three hosts are connected through a Gb switched VLAN, and we monitor their communication by running `tcpdump` on a span port on a separate host (not shown). We configure our setup to only permit updates to be sent from the generator to the IRSCP server and from there to the receiver.

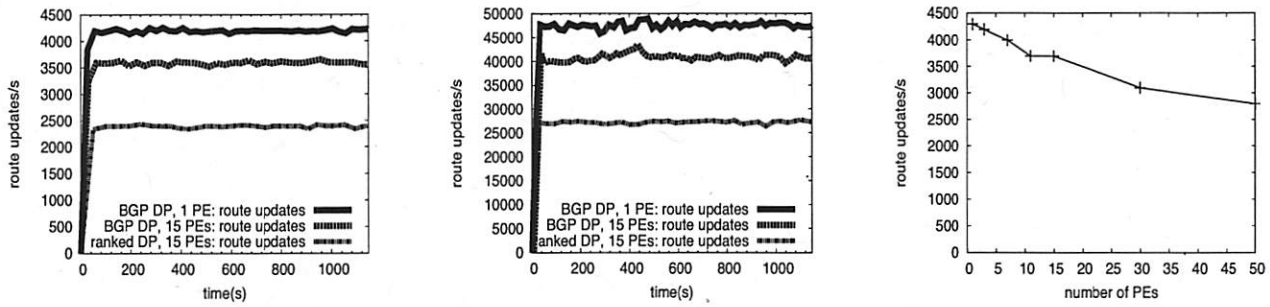


Figure 9: IRSCP server throughput. (a) Sustained input rate: from update generator to IRSCP server. (b) Sustained output rate: from IRSCP server to update receiver. (c) Input rate for varying number of PEs sustained for 40 seconds.

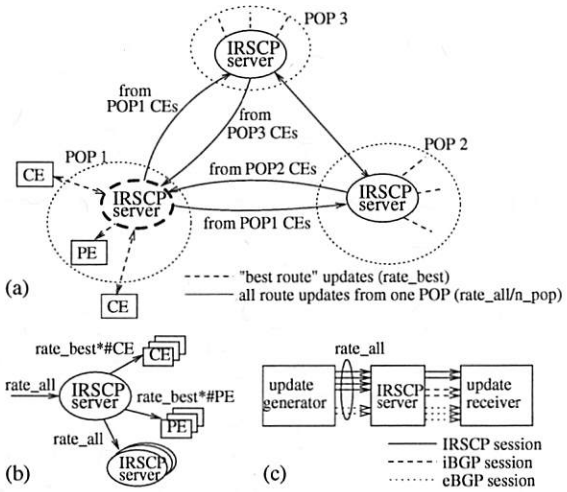


Figure 10: (a) Communication peers of IRSCP server under test. (b) Update rates into and out of the IRSCP server. (c) Experimental setup.

After being loaded with a routing table of 234,000 prefixes, the update generator randomly picks prefixes from the routing table to produce the updates. To ensure that the IRSCP server's RIB contains a sufficient number of routes per prefix to choose from, the update generator and the IRSCP server maintain 26 sessions. (This number is based on a highly conservative estimate of the average number of routes per prefix in the Tier-1 ISP [30].) Each time the update generator sends an update message it picks a different session through which to send the message (using round-robin). The message modifies a BGP attribute (the "aggregator" attribute) of the prefixes contained in the message.

The IRSCP server and the update receiver maintain 40 IRSCP sessions, 15 iBGP sessions, and 240 eBGP sessions, reflecting an ISP network consisting of 40 POPs, each of which contains 15 PEs and has sessions with 240 CEs. By virtue of route selection in the IRSCP server each output BGP session receives a 1/26 fraction of $rate_{all}$, which turns out to correspond to $rate_{best}$ [30]. We configure one of the 26 input update-generator sessions as an eBGP session and the remainder as IRSCP sessions, ensuring a rate of $rate_{all}/26 > rate_{all}/n_{pop}$ on each output IRSCP session. Finally we instrument the IRSCP server and update generator to send at most three route updates per update message, reflecting the average number of updates per message observed at a route collector in the Tier-1 ISP's network.

5.3 Throughput

We determine the maximum value of input rate ($rate_{all}$) that the IRSCP server can sustain for an extended period of time, as follows. To discover the maximum input rate, we gradually increase the input rate and compare the observed output rate with an expected output rate of $rate_{all}/26 \cdot (n_{irscp} + n_{ibgp} + n_{ebgp})$. When the input rate is below its maximum, the output rate corresponds to the expected output rate. Once the input rate exceeds its maximum, the output rate steadily declines with increasing input rate.

Using this procedure we compare the performance of the standard BGP decision process with the explicitly ranked decision process, and evaluate the impact of the per-PE decision process. For the standard BGP decision process we find a maximum $rate_{all} \approx 3600$ updates/s, corresponding to an expected output rate of 40,846 updates/s. Figures 9(a) and (b) (*BGP DP, 15 PEs*) show the observed input and output rates during a run lasting twenty minutes, averaged over 30-second intervals. While the output rate is not as stable as the input rate, on average it corresponds to the expected output rate, and the figure shows that it is sustainable. Next, we load explicit rankings for 60,000 prefixes, each listing 26 egress IDs in some arbitrary order. In this case we find a maximum $rate_{all} \approx 2400$ updates/s, corresponding to an expected output rate of 27,231 updates/s. As expected, the explicitly ranked decision process is slower than the BGP decision process, since it runs in time quadratic rather than linear in the number of routes per prefix. Again, Fig-

ure 9 (*ranked DP, 15 PEs*) shows that the IRSCP server can sustain this workload.

Finally, to evaluate the impact of the per-PE decision process (vs. a single, router-wide decision process), we run an experiment again based on the BGP decision process, but in which all but one of the iBGP sessions between the IRSCP server and the update receiver are replaced by an eBGP session. In this case we find that the IRSCP server sustains a maximum $rate_{all} \approx 4200$ updates/s and produces the expected output rate of 47,654 updates/s (*BGP DP, 1 PE* in Figure 9). Figure 9(c) plots $rate_{all}$ for a varying number of PEs (and using the BGP decision process), but evaluated by sustaining the rate for only 40 seconds, rather than 20 minutes.

While the sustained throughputs are less than our maximum measurement-based estimate for $rate_{all}$ of 4900 updates/s, the IRSCP server easily manages to keep up with our 95th percentile estimate of 600 updates/s in all experiments, even when increasing the number of PEs to 50. Hence, we expect that an improved implementation of flow control in the IRSCP server should sufficiently slow down senders during times that the offered load exceeds the maximum sustainable throughput.

5.4 Memory consumption

We evaluate memory usage of an IRSCP server as a function of the number of routes it stores. We use a subset of the setup in Figure 10(c): the IRSCP server under test and the update generator. Also in this experiment we add BGP attributes from a routing table in the ISP network from October 2006. We vary the number of sessions between the update generator and the IRSCP server from 0 to 20 and measure the memory usage of the IRSCP server's RIB. We find a linear increase from 18.2 MB (0 sessions) to 226 MB.

Next, we examine memory usage of the explicit rankings. We configure an IRSCP server with rankings for 15 PEs and vary the number of ranked prefixes from 0 to 130,000 (but without loading their routes). Each ranking for a prefix and PE consists of 26 egress IDs. We measure the process size of `openbgpd`'s route decision engine, which stores the rankings. Again we find a linear increase from 840 KB (0 prefixes) to 994 MB (130,000 prefixes). Our current implementation is not optimized for space allocation, and, as a result cannot store rankings of this size for more than 130,000 prefixes. After we load 26 copies of the routing table, our prototype supports rankings for up to 75,000 prefixes on our test machines. However, we expect 26 egress IDs per prefix to be vastly more than needed in practice; five egress IDs per prefix can be stored for all 234,000 prefixes with full routing tables loaded.

6 Related work

IRSCP extends our previous work on route control architectures [6, 12, 29] in three important ways. First, we introduce a modified BGP decision process which we expose to route control applications. Second, IRSCP has *full visibility* of all routes available to the network through its eBGP interaction with neighboring networks. This is in contrast to a phase-one, iBGP-only RCP [6], where routers in the IRSCP-enabled network only pass *selected* routes to IRSCP. Having full route visibility prevents route oscillations within the network [3, 15, 21] and simplifies route control applications. Third, IRSCP distributes the route selection functionality, whereas the simple server replication presented in [6] required each replica to scale to accommodate the entire network. Bonaventure *et al.* [5] propose sophisticated route reflectors but limit their changes to the iBGP infrastructure. The 4D project proposes a refactoring of the network architecture, creating a logically centralized control plane separate from forwarding elements [14, 32].

The IETF ForCES working group examines a separation of control plane and forwarding plane functions in IP network elements [19]. This work has a much narrower focus on defining the interface between control and forwarding elements, without considering interaction between different control plane elements. Once ForCES-enabled routers become available, IRSCP's iBGP communication with local routers might conceivably be replaced by a ForCES protocol.

An earlier IETF proposal [10, 25] provides limited route control by defining a new BGP attribute subtype, the cost community, which can be assigned to routes and used to break ties at a certain "point of insertion" in the BGP decision process. This proposal does not indicate under what conditions the cost community would be safe to use; by contrast, we show how our rankings should be constrained to ensure consistency.

7 Conclusion

The ultimate success of a logically centralized control plane architecture will depend not only on its ability to enable new functionality, but also on its ability to provide a scalable and robust routing function to large and growing provider networks. We address each of these points by presenting a distributed realization of the Intelligent Route Service Control Point (IRSCP) that partitions work between instances and allows redundancy requirements to drive the extent to which the system is replicated. We also move beyond BGP capabilities by allowing route control applications to directly influence the route selection process by providing a ranking of

egress links on a per-destination and per-router basis. We demonstrate the utility of this change with a simple load-balancing application. As future work, we plan to investigate to what extent an IRSCP that has complete control and visibility allows a simplification of the expression and management of conventional routing policies.

Acknowledgments. We would like to thank Yuvraj Agarwal, Michelle Panik, Barath Raghavan, Rangarajan Vasudevan, Michael Vrable and the anonymous reviewers for their helpful comments on previous versions of the paper. This work was supported in part by the National Science Foundation through grant CNS-0347949.

Notes

[1] Following our previous taxonomy [12], the architecture presented in this paper therefore represents a phase-two RCP.

[2] A similar problem is introduced by *cold-potato* routing based on the BGP MED attribute (Step 4 in Table 1). We omit the IRSCP solution to this problem for lack of space.

[3] MED oscillation continues to be observed in the Internet [31].

[4] However, if a black-hole egress ID is present in the ranking, then IRSCP still excludes egress routes that are ranked below the black-hole egress route before executing the hot-potato steps.

[5] Ignoring the fact that eBGP export policies may still be applied.

[6] By "update" we mean a route update, rather than an update *message*, which may contain several route updates.

[7] The machines are actually dual-processor, but OpenBSD uses only one of the processors.

References

[1] Aditya Akella, Jeffrey Pang, Anees Shaikh, Bruce Maggs, and Srinivasan Seshan. A Comparison of Overlay Routing and Multihoming Route Control. In *ACM SIGCOMM*, September 2004.

[2] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *ACM SOSP*, pages 131–145, October 2001.

[3] Anindya Basu, Chih-Hao Luke Ong, April Rasala, F. Bruce Shepherd, and Gordon Wilfong. Route Oscillations in I-BGP with Route Reflection. In *ACM SIGCOMM*, pages 235–247, 2002.

[4] T. Bates, R. Chandra, and E. Chen. BGP Route Reflection - An Alternative to Full Mesh IBGP. RFC 2796, April 2000.

[5] O. Bonaventure, S. Uhlig, and B. Quoitin. The case for more versatile BGP Route Reflectors. Internet draft: draft-bonaventure-bgp-route-reflectors-00.txt, July 2004.

[6] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a Routing Control Platform. In *ACM/USENIX NSDI*, 2005.

[7] Matthew Caesar and Jennifer Rexford. BGP routing policies in ISP networks. IEEE Network, November 2005.

[8] Kuan-Ta Chen, Chun-Ying Huang, Polly Huang, and Chin-Luang Lei. Quantifying Skype User Satisfaction. ACM SIGCOMM, September 2006.

[9] Kuan-Ta Chen, Polly Huang, Guo-Shiuan Wang, Chun-Ying Huang, and Chin-Luang Lei. On the Sensitivity of Online Game Playing Time to Network QoS. IEEE Infocom, April 2006.

[10] Cisco Systems. BGP Cost Community. Cisco IOS Documentation.

[11] Nick Duffield, Kartik Gopalan, Michael R. Hines, Aman Shaikh, and Jacobus E. van der Merwe. Measurement Informed Route Selection. Passive and Active Measurement Conference, April 2007. Extended abstract.

[12] Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Jacobus E. van der Merwe. The Case for Separating Routing from Routers. ACM SIGCOMM FDNA, Aug 2004.

[13] Nick Feamster and Jennifer Rexford. A Model of BGP Routing for Network Engineering. In *SIGMETRICS*, June 2004.

[14] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM CCR*, 35(5), 2005.

[15] Timothy Griffin and Gordon T. Wilfong. Analysis of the MED Oscillation Problem in BGP. In *ICNP*, 2002.

[16] Timothy G. Griffin and Gordon Wilfong. On the Correctness of IBGP Configuration. In *ACM SIGCOMM*, pages 17–29, New York, NY, USA, 2002. ACM Press.

[17] Geoff Huston. Wither Routing? The ISP Column, November 2006.

[18] Josh Karlin, Stephanie Forrest, and Jennifer Rexford. Pretty Good BGP: Improving BGP by Cautiously Adopting Routes. In *ICNP*, November 2006.

[19] H. Khosravi and T. Anderson. Requirements for Separation of IP Control and Forwarding. IETF RFC 3654, November 2003.

[20] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding BGP Misconfiguration. *SIGCOMM Comput. Commun. Rev.*, 32(4):3–16, 2002.

[21] D. McPherson, V. Gill, D. Walton, and A. Retana. Border Gateway Protocol (BGP) Persistent Route Oscillation Condition. RFC 3345, August 2002.

[22] James Nichols and Mark Claypool. The Effects of Latency on Online Madden NFL Football. In *ACM NOSSDAV*, June 2004.

[23] Nathan Patrick, Tom Scholl, Aman Shaikh, and Richard Steenbergen. Peering Dragnet: Anti-Social Behavior Amongst Peers, and What You can Do About It. NANOG 38, October 2006.

[24] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). IETF RFC 4271, 2006.

[25] Alvaro Retana and Russ White. BGP Custom Decision Process. Internet draft: draft-retana-bgp-custom-decision-00.txt, October 2002.

[26] Stefan Savage, Andy Collins, Eric Hoffman, John Snell, and Thomas Anderson. The End-to-End Effects of Internet Path Selection. In *ACM SIGCOMM*, September 1999.

[27] Renata Teixeira, Timothy G. Griffin, Mauricio G. C. Resende, and Jennifer Rexford. TIE Breaking: Tunable Interdomain Egress Selection. In *CoNEXT*, 2005.

[28] Renata Teixeira, Aman Shaikh, Tim Griffin, and Jennifer Rexford. Dynamics of Hot-Potato Routing in IP Networks. In *ACM SIGMETRICS*, 2004.

[29] Jacobus E. van der Merwe et al. Dynamic Connectivity Management with an Intelligent Route Service Control Point. ACM SIGCOMM INM, October 2006.

[30] Patrick Verkaik, Dan Pei, Tom Scholl, Aman Shaikh, Alex C. Snoeren, and Jacobus van der Merwe. Wresting Control from BGP: Scalable Fine-grained Route Control. Technical report, AT&T Labs-Research, January 2007.

[31] J. Wu, Z. Mao, J. Rexford, and J. Wang. Finding a Needle in a Haystack: Pinpointing Significant BGP Routing Changes in an IP Network. In *USENIX NSDI*, 2005.

[32] Hong Yan, David A. Maltz, T. S. Eugene Ng, Hemant Gogineni, Hui Zhang, and Zheng Cai. Tesseract: A 4D Network Control Plane. In *ACM/USENIX NSDI*, 2007.

A Comparison of Structured and Unstructured P2P Approaches to Heterogeneous Random Peer Selection *

Vivek Vishnumurthy and Paul Francis
Department of Computer Science, Cornell University, Ithaca, NY 14853
{vivi,francis}@cs.cornell.edu

Abstract

Random peer selection is used by numerous P2P applications; examples include application-level multicast, unstructured file sharing, and network location mapping. In most of these applications, support for a heterogeneous capacity distribution among nodes is desirable: in other words, nodes with higher capacity should be selected proportionally more often.

Random peer selection can be performed over both structured and unstructured graphs. This paper compares these two basic approaches using a candidate example from each approach. For unstructured heterogeneous random peer selection, we use Swaplinks, from our previous work. For the structured approach, we use the Bamboo DHT adapted to heterogeneous selection using our extensions to the item-balancing technique by Karger and Ruhl. Testing the two approaches over graphs of 1000 nodes and a range of network churn levels and heterogeneity distributions, we show that Swaplinks is the superior random selection approach: (i) Swaplinks enables more accurate random selection than does the structured approach in the presence of churn, and (ii) The structured approach is sensitive to a number of hard-to-set tuning knobs that affect performance, whereas Swaplinks is essentially free of such knobs.

1 Introduction

A number of P2P or overlay applications need to select random peers from the P2P network as part of their operation. A simple but poorly scaling way to do random peer selection is to disseminate a list of all nodes to all nodes. To randomly select another node, each node simply selects randomly from its list. Early gossip protocols that needed uniform random peer selection typically assumed this approach. A scalable alternative approach is to build a sparse graph among the peers, and then use some kind of walk through the graph to do random node selection.

An early example of this approach was an overlay multicast protocol called Yoid [1]. Yoid constructed a random graph over which random walks would be used to discover nodes that might be included in a multicast tree. More recent overlay multicast protocols that utilize random node selection include Bullet [2] and Chainsaw [3].

Gnutella-style unstructured file sharing networks utilize a random selection component when searching for nodes having desired files. To improve the scalability of this file search, GIA [4] proposes using random walks rather than flooding.

Random node selection also plays an important role in proximity addressing schemes like Vivaldi [5] and PALM [6]: these schemes need to select random peers in the network to measure their latencies from the peers and compute their coordinates.

In many of these examples, it is important that random selection follow a given non-uniform probability distribution. In other words, some nodes are to be selected with higher probability than others. The primary reason for this is to accommodate nodes with differing capacities. We refer to this problem of selecting each node with probability proportional to its specified capacity as *heterogeneous random selection*. The need to accommodate heterogeneity is especially acute for file searching in Gnutella-like file sharing networks, as the use of supernodes attests. The primary focus of GIA is this heterogeneity in unstructured networks. Accommodating node heterogeneity is also important in overlay multicast algorithms [7, 8, 9, 10].

Given the number and variety of P2P and overlay applications that use random node selection, in previous work [11], the authors designed Swaplinks, a general-purpose unstructured P2P algorithm to provide a heterogeneous random node selection *primitive* that could be used by a wide range of P2P and overlay applications. Swaplinks builds a random graph in which each node's degree is proportional to its desired degree of heterogene-

*This material is based upon work supported by the National Science Foundation under Grant No. 0338750. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF).

ity, and then uses random walks over that graph to do node selection. The previous work showed, using simulations, that Swaplinks was the most attractive unstructured random selection technique: It gives fine-grained control over both the probability that a node is selected and the overhead seen at each node. It is efficient, scalable, robust (to churn, and to wide variations in node capacities), and simple. In this paper, we implement Swaplinks, and provide a comprehensive evaluation of the Swaplinks implementation, thus validating the previous simulation results.

Heterogeneous random selection in the example applications cited earlier can also be potentially realized by structured approaches. Our previous work, however, examined only unstructured approaches to random selection, because of our intuition that they would be simpler than structured approaches, and that this simplicity would ultimately lead to a more scalable and robust system. A primary goal of this paper is to test our intuition about the relative simplicity of structured and unstructured approaches to heterogeneous random selection through a performance comparison between the two approaches. We choose the "item-balancing" algorithm by Karger and Ruhl for load balancing in structured P2P networks [12] as the basis for the structured random selection approach, and we use Swaplinks as the unstructured approach. The basic idea in using the item-balancing algorithm in our setting is to assign identifiers in the DHT number space such that a larger portion of the number space maps proportionally into high-capacity nodes, and a smaller portion maps into low-capacity nodes. High capacity nodes, by virtue of "owning" a larger portion of the number space, will be selected proportionally more often by queries issued to uniformly random identifiers. We implement the Karger/Ruhl approach over the Bamboo DHT [13], and call this approach KRB. We chose Bamboo because it is a stable well-maintained open software for DHTs, and because it is a second generation DHT, designed using the best principles from the earlier, first generation DHTs (like Chord [14] and Pastry [15]). This minimizes the chances that the results are an artifact of a poor DHT implementation.

The performance comparison between KRB and Swaplinks shows that KRB performs less well in the face of churn, and has a number of hard-to-set tuning knobs that affect performance. While we need more comparisons (with other structured approaches) to be certain of this, the performance comparison goes a long way toward validating our intuitive concern about the relative complexity of using DHTs for heterogeneous peer selection.

Overall, this paper makes two contributions:

- We implement an open source library [16] that provides heterogeneous unstructured random graph construction and random node selection primitives based on Swaplinks. We also measure the performance of the Swaplinks implementation for both random graph construction and random selection, and in so doing validate earlier simulation results.
- We modify the Karger/Ruhl load balancing algorithm for heterogeneous random peer selection, and compare its performance as a random selection mechanism with that of Swaplinks.

We next describe related work in Section 2. We describe the Swaplinks algorithm and its implementation in Sections 3 and 4, and the KRB method in Section 5. In Section 6 we give a performance evaluation and comparison of both algorithms. Finally, we discuss issues and future work in Section 7.

2 Related Work

2.1 Structured P2P Networks

All structured P2P systems modeled as DHTs (e.g. [17, 14, 15], etc.) assign identifiers to nodes, typically at random. Random selection in DHTs can be done by randomly choosing a value from the DHT number space, and routing to that value. The problem of random node selection in DHTs, then, boils down to the problem of assigning identifiers appropriately.

Even where uniform random selection is desired, assigning a single random identifier to each node is inadequate, because any non-uniformities in the random assignments persist over time. Consistent hashing schemes deal with this by assigning multiple random identifiers[18], and DHTs have proposed something similar, namely creating multiple virtual replicas of each node in the DHT. To achieve heterogeneity, each node is replicated a number of times proportional to its capacity ([19, 20]). This approach however entails a blowup in network and computational overheads, and so is not an attractive approach.

A modified multiple virtual node approach is used in Y_0 [21]. Here, virtual node identifiers for each node are selected from a small range of identifiers; the authors utilize the proximity of the node's identifiers to avoid having to maintain separate routing entries for each virtual node. While this scheme is interesting, and a potential candidate for comparison, it has not been analyzed or tested for robustness to high churn. Y_0 also needs all nodes to know (at least roughly) the number of nodes in the system, which might be an issue under high churn.

Ledlie and Seltzer [22] present the *k-choices* algorithm for load balancing in settings with skewed query distributions and heterogeneous capacities. *k-choices* is similar to KRB, in that both place nodes at IDs that minimize

load imbalance. The difference is that *k-choices* assumes that each node knows its absolute desired load, whereas in KRB, nodes only have a notion of relative desired load.

Accordion [23] and HeteroPastry [24] give schemes that tailor nodes' degrees and their message loads according to capacity and network activity. These schemes however do not provide capacity dependent namespace partitioning, and so cannot support heterogeneous random selection by routing to uniformly randomly selected IDs. An alternative approach might have been to use unbiased random walks over these networks for random selection, but the control over degrees in these schemes is not fine-grained enough (i.e., average node degrees are not proportional to capacities) for this to result in the desired selection distribution[1].

Karger and Ruhl propose two schemes in their papers for load balancing in DHTs [12, 25]. The first results in a constant factor bound on ID spaces between successive nodes, but cannot handle the case where the ID spaces are to be split according to capacities. The second scheme looks at item load balancing, where the number of items that are stored at any node should be within bounds and dependent on node capacity. With minor variations, we could modify this scheme to split ID space according to node capacities and run over Bamboo – we call this KRB. We use KRB as the candidate structured approach for our performance comparisons.

2.2 Unstructured P2P Networks

In previous work [11], we found Swaplinks to be the best algorithm for constructing unstructured P2P graphs suitable for heterogeneous random selection. Here is a brief overview of other unstructured approaches.

GIA extends Gnutella by making both graph-construction and query-resolution sensitive to node capacities [4]. High-capacity nodes here have higher degrees, and are more likely to be traversed by random walks. While Swaplinks shares these two features with GIA, Swaplinks exhibits more accurate control over degree and probability of selection. Other examples of unstructured graph construction schemes include Araneola [26], an approach by Law and Siu [27], and SCAMP [28]. None of these take node heterogeneity into account.

The Ransub [29] mechanism can be used as a random node selection primitive, but as was the case with the previously mentioned schemes, does not take into account node heterogeneity. The Metropolis-Hastings algorithm [30] and the Iterative-Scaling algorithm [31] can be used to achieve desired probabilities of selection over any underlying graph. But when the underlying graph has node degrees close to the desired probabilities, like Swaplinks does, random selection primitives achieve the desired distribution much more efficiently (e.g., random walks need to take far fewer hops).

3 The Swaplinks Algorithm

Our random selection API consists of the following core procedures:

- $join(numLinks)$
- $node = select()$
- $listOfNodes = listNeighbors(callBack)$

The $join()$ procedure causes the joining node to establish random links with other, already joined nodes. The parameter $numLinks$ indicates how many neighbors the joining node should try to obtain. On average a node will end up with twice as many neighbors as the value $numLinks$. This is because other nodes will in turn select a given node as their neighbor.

The value of $numLinks$ is set to be proportional to the probability with which the node should be selected. For instance, if a node A should be selected with twice the probability of node B, then node A will set $numLinks$ to be twice that of node B. It is up to the application to know what values to choose for $numLinks$. Typically an application would choose a value of $numLinks = 3$ for its lowest capacity nodes, and select values proportionally higher for higher capacity nodes. The value 3 is chosen as the minimum to insure that even the lowest capacity node has a low probability of partition from the rest of the network. Higher values reduce the probability even more.

When a node wishes to randomly select another node, it calls $select()$. This causes a random walk to be taken through the random graph. The number of hops in the walk is a fixed value, 10 by default. The node at which the walk ends is the selected node. The value $numLinks$ plays two important roles here. First and foremost, the Swaplinks design ensures that the walk will end at nodes with higher $numLinks$ values with proportionally higher probability. Second, nodes with a higher $numLinks$ value will serve as intermediate hops in walks with higher probability. This second effect results in the load required to participate in the algorithm by any given node to also be proportional to its capacity. There are a number of possible variations on the $select()$ call: for instance the length of the walk may be specified, or the identity of all nodes traversed during the walk may be returned.

Some P2P applications may simply wish to use the underlying graph directly. For instance, a BitTorrent might use the neighbors selected by the $join()$ procedure as the nodes with which it exchanges file blocks. The $listNeighbors(callBack)$ procedure allows this. In

addition to providing the current set of neighbors, a callback routine allows Swaplinks to inform the application whenever the neighbor set has changed.

In building and maintaining a random graph, each node labels each of its links to a neighbor node as either an outlink or an inlink. These labels have nothing to do with the direction messages may pass over them: messages may pass in both directions. Rather, the label is chosen based on which node initiated creation of the link. The node that initiated the link labels it an outlink, and other node labels it an inlink. Correspondingly, neighbor nodes are labeled as out-neighbors or in-neighbors. The outdegree is the number of outlinks, and the indegree is the number of inlinks. Every link is an outlink in one direction and an inlink in the other.

Likewise, there are two types of fixed-length random walks:

OnlyInLinks: The walk is forwarded to a randomly chosen in-neighbor.

OnlyOutLinks: The walk is forwarded to a randomly chosen out-neighbor.

Every node always maintains an outdegree of $numLinks$, by finding $numLinks$ out-neighbors when it first joins, and by replacing any out-neighbor it loses with another one. This is done in such a way that nodes tend to have the same number of inlinks as outlinks, though they may have slightly greater or fewer than $numLinks$ inlinks.

There are three cases the algorithm must cover:

1. A joining node is adding selected out-neighbors
2. A node is replacing a lost out-neighbor
3. A node is replacing a lost in-neighbor

To find a new out-neighbor for the first case, a joining node (say A) initiates a fixed length OnlyInLinks walk from one of its entry nodes. The node (say B) where the walk ends is chosen as an out-neighbor for the new node. Node B then randomly selects one of its in-neighbors C, and "gives" that in-neighbor to A. In other words, C loses B as an out-neighbor, and gains A as an out-neighbor.

The result of this transaction is that A gains both an out-neighbor (B) and an in-neighbor (C). After A is done finding all of its $numLinks$ out-neighbors, it will also have an equivalent number of in-neighbors. Node B will have gained one in-neighbor (A) and lost another (C), so it comes out even. Node C will have lost one out-neighbor (B) and gained another (A), so it also comes out even.

If a node (say A) loses an existing out-neighbor (the second case above), it likewise takes an OnlyInLinks walk, and creates an outlink with the discovered node (say B). However, in this case, B does not give one of its in-neighbors to A. Rather, B ends up with an extra in-neighbor. Had B given A an in-neighbor, then A would have ended up with an extra in-neighbor instead.

Finally, if a node (A) loses an existing in-neighbor (the third case above), it sees if its number of in-links is less than its $numLinks$. If so, it takes an OnlyOutLinks walk. The node (B) discovered by the walk then donates one of its in-neighbors to A if B's number of inlinks is greater than half its $numLinks$.

The above modes of link-formation could lead to the creation of multiple links between the same pair of neighbors; Swaplinks makes no effort to eliminate these multiple links. This makes dealing with very small networks straightforward.

The rationale behind using the OnlyInLinks and OnlyOutLinks walks in Swaplinks is as follows: The OnlyInLinks walk selects each node with a probability roughly proportional to its outdegree. The OnlyOutLinks walk, on the other hand, selects each node with probability roughly proportional to its indegree. Thus Swaplinks, by using the OnlyInLinks walk, ensures that the load placed on each node is proportional to its outdegree. And by employing OnlyOutLinks to deliberately look for inlinks in the presence of churn, it tends to find nodes with disproportionately large indegrees, thus stealing the surplus inlinks from such nodes and ensuring that nodes' indegrees stay close to their outdegrees.

In this paper, application-requested node selection ($select()$) uses OnlyInLinks walks, as opposed to the other random walks tested in [11]. While both OnlyInLinks and the random selection walks in [11] result in selection proportional to nodes' outdegrees, OnlyInLinks is simpler and thus the more attractive method to use.

Simulations in [11] show that Swaplinks builds graphs where the degree distribution closely resembles the desired distribution. The graphs scale well to large sizes, and lend themselves well to random peer selection. The resultant message load on nodes and the frequency of selection vary linearly with the degree. Swaplinks implementation results presented later in this paper corroborate these findings.

A feature of Swaplinks that makes it attractive from a practical viewpoint is that it is free of "tuning knobs": It has no parameters to set, apart from the neighbor heartbeat frequency parameter (present in most distributed systems). We avoid having to tune the hop-length for different random walks by making all walks 10 hops in length, which is a conservatively large value.

4 Swaplinks Implementation

Our system is implemented in C++ on Linux. We use TCP sockets for neighbor connections. Each node sends heart-beat messages to each of its neighbors every 2 sec-

onds, and assumes that a neighbor is dead if it does not receive a heart-beat from it for 10 seconds. [2]

A newly entering node initiates the required number of neighbor discovery walks, restricting the number of outstanding neighbor walks to 10 at any time. A neighbor discovery walk is re-attempted if it fails to return an appropriate neighbor within a period of 2 seconds.

We currently have an implementation of a *rendezvous server* that helps new nodes join the system. The rendezvous server remembers a small number (currently 10) of the most recently joined nodes, and newly joining nodes use these nodes to start their neighbor discovery walks. This rendezvous mechanism is light-weight, and makes sure no single node is overloaded with the responsibility of helping new nodes join the network. The rendezvous mechanism could be made more robust by also having the rendezvous server remember a small number of random other nodes in the network, by periodically taking random walks, or by having newly joined nodes report one or two of their neighbors.

The application using Swaplinks communicates with the Swaplinks module via a TCP socket. Swaplinks exports the API described in section 3 to the application over the socket by using appropriate serialization.

One application has currently been implemented over Swaplinks, namely a heterogeneous overlay multicast protocol called ChunkySpread [7] that uses Swaplinks to both construct a heterogeneous random graph and do random peer selection. Each Chunkyspread node is involved in multicast data transmission (and reception) with multiple other nodes; this set of peers is a subset of the set of the neighbors in the Swaplinks graph. A small set of ChunkySpread nodes (the nodes that originate the multicast stream) need to discover an additional set of peers. This is done using Swaplinks peer selection. In addition to ChunkySpread, the Swaplinks algorithm is being used in other applications under current development, like the NUTSS toolkit for NAT traversal in P2P systems [32], and a P2P file backup system.

We are also currently experimenting with an alternate heart-beat mechanism, called *smart-pinging*, that reduces heart-beat load at nodes with very high degrees. We describe smart-pinging and give a preliminary evaluation of the technique in Section 6.4.

5 Adapting Bamboo to Heterogeneity

Performing random selection on a DHT, with no regards to heterogeneity, and assuming the ID space is apportioned uniformly among all nodes, is simple: pick a uniformly random ID in the ID space, issue a random selection query to that ID, and select the node where the query ends. For this simple querying mechanism to still be applicable when there are differences in node capacities, we need to split nodes' ID spaces in proportion to their capacities (where a "node's ID-space" denotes the extent of ID-space that the node owns). For a simpler design of the heterogeneous random selection scheme, we choose to compute a node's ID space as the space between its successor in the ring and itself. We discuss how we simulate this feature in Bamboo later in this section.

To achieve capacity-dependent ID space allocation, we develop a scheme based on the item-balancing algorithm (henceforth referred to as *K-R*) presented in [12, 25]. Nodes in K-R periodically send messages to one another, and share loads when a load imbalance is perceived. The item-balancing algorithm in [12] performs load sharing through movement of nodes to new IDs, but does not address the issue of heterogeneity, whereas the one in [25] takes heterogeneity into account, but does load sharing by transferring *items* from heavily loaded nodes to lightly loaded ones. Our scenario is slightly different from either of the above two, since we need nodes to move to new IDs so as to do ID space partitioning, and we need this partitioning to be capacity sensitive.

We now outline KRB, our adaptation of the K-R algorithm. The basic aim of KRB is to even out the *relative* loads of all nodes, where a node's relative load is its ID space load divided by its capacity. As in K-R, each node periodically sends out a message to a randomly chosen ID, embedding its load information – we call such messages "KRB load messages". Noting that a node's moving to a new ID can affect the ID spaces of (up to) 3 nodes (the moving node, the moving node's old predecessor, and the moving node's new predecessor), in KRB, we examine the change in load at *all* nodes whose loads are affected by the move. This is an extension of K-R, where the loads at only the moving node and the moving node's new predecessor are examined. If we examined the loads at only these two nodes, it would be possible for a huge load to be inadvertently dumped on the unconsidered third node (the moving node's old predecessor) as a result of the move; by considering all the three nodes, we avoid this possibility.

Looking at a single KRB load message, let us denote by S the node that sent out the message, by R the node that receives the message, and by P the predecessor of R. Now R decides if it should move to share S's ID-space, based on the value of the *objective function*, computed as follows:

$$r = \frac{L_R + L_S + L_P}{C_R + C_S + C_P}$$

$$ObjFn(R,S,P) = \Sigma_{N \in \{R,S,P\}} \left| \frac{L_N}{C_N} - r \right| \quad (1)$$

where L_N is node N's ID-space load, which is equal to the space between N and its successor, and C_N is node N's capacity.

If R were to move, it would move to ID R' such that

$$L_{R'} = \frac{L_S \cdot C_R}{C_R + C_S} \quad (2)$$

That is, the new ID is the one that splits the space between S and its successor in direct proportion to their capacities. If R were to move to R', the objective function would take on a new value, computed similarly to above. Finally, R does make the move if the objective function value reduces by more than a threshold ratio (called the *KRB-threshold*, set to 0.2).

The computation of the objective function above can be seen as a greedy step taken towards minimizing the system-wide objective function, given below.

$$r_{all} = \frac{\Sigma_{N\ in\ system}(L_N)}{\Sigma_{N\ in\ system}(C_N)}$$

$$ObjFn(overall) = \Sigma_{N\ in\ system}\left|\frac{L_N}{C_N} - r_{all}\right| \quad (3)$$

Since individual nodes do not know the value of r_{all}, they use local knowledge to compute r as shown above as an estimate.

The above description assumes that the node that sent the initial message S is not already the predecessor of the node that receives the message R. If R does happen to be the successor of S, there is no other third node whose load will be affected if R were to move to any point in between S and its successor. So now R moves if the following condition holds:

$$\frac{L_S}{C_S} < \epsilon \frac{L_R}{C_R} \quad OR \quad \frac{L_R}{C_R} < \epsilon \frac{L_S}{C_S}$$

where we set ϵ to 0.8. This criterion is identical to the one used in K-R.

Simulating a node's ID-space in Bamboo:
To make this scheme work in Bamboo, we need to make sure that the probability that a node is selected is proportional to the ID space for which it is the closest predecessor. However, in Bamboo, a query is routed to the node numerically closest to the destination, rather than to the closest predecessor. So when a node receives a random selection query, it examines the intended destination ID and forwards it to the immediate predecessor of that ID.

Our primary goal in adapting the Karger/Ruhl scheme to Bamboo was capacity-sensitive random peer selection. Admittedly, this scheme does not balance message load according to capacities (during the construction of the KRB network or during random selection), as we only tailor nodes' ID spaces, and not their routing tables. Accordingly, in this paper, we evaluate KRB as a heterogeneous selection mechanism alone, and do not place emphasis on the message load distribution that occurs while constructing the KRB P2P network. Schemes that reactively tailor the neighborhood size based on capacity, such as those proposed in Accordion [23] and HeteroPastry [24] could be used with KRB to achieve both capacity-sensitive probability of selection and capacity-sensitive message load distribution during graph construction.

6 Performance Evaluation

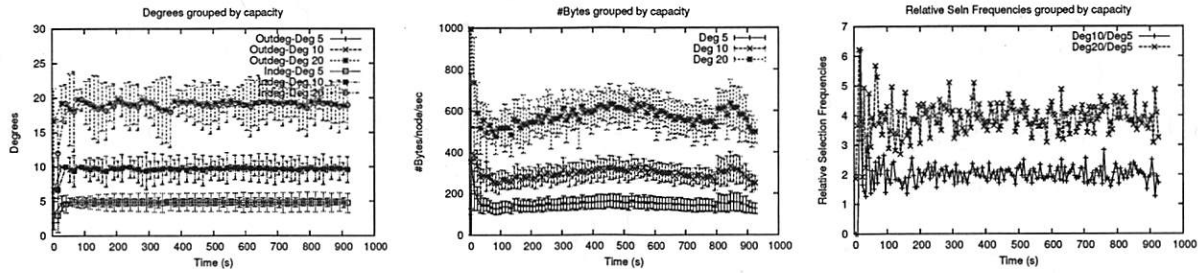
We test Swaplinks through an emulation of a 1000 node network on either a local (Cornell) cluster of 5 machines with 4 CPU's each, or a 20 CPU cluster on Emulab. We achieve this size by launching a number of processes that in turn launch the required number of individual instances of our system. We preserve the semantics of communication here: all communication still takes place through sockets. The CPU loads here were mostly small enough to be negligible as a factor in the results. We also test the same implementation on PlanetLab.

For the emulation, we use a Transit-stub [33] topology consisting of 100 routers to mimic latencies between peers. Each peer picks a stub router uniformly at random. All messages to be sent are buffered at the sender for the appropriate amount of time (computed as a function of the stub routers of the source and destination). We also add jitter as a random value that ranges between 0 and 25% of the end-to-end latency.

Launching KRB networks of a similar size (500-1000 nodes) by multiplexing several instances on single hosts on local clusters proved infeasible because of high CPU load factors due to the Bamboo implementation. We instead evaluate KRB using the simulator available with Bamboo's standard code distribution. We use the same Transit-Stub topology as earlier to calculate message delays in the KRB network. We had to restrict our comparisons to 1000 node networks as the Bamboo simulator, with our modifications, consumes too much memory for larger sizes.

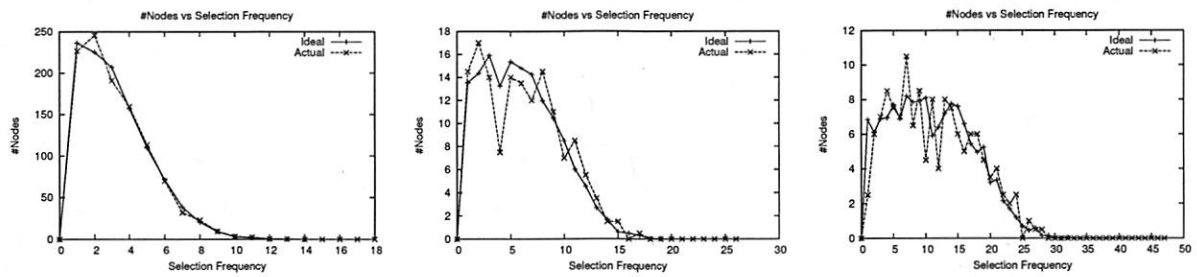
All KRB nodes use a single entry node (called a *gateway* node by Bamboo) when they first enter the system. A node that leaves its present spot and rejoins the system as part of the KRB ID space readjustment scheme uses the set of neighbors it had before it left the system as its gateway nodes.

We now give a road map of the experimental results that we will be presenting in the subsequent portions of the paper. We first test 1000 node networks of both Swaplinks and KRB under two different representative



(a) Degree Distribution (b) Message Loads (c) Relative Selection Frequencies

Figure 1: Swaplinks under high churn and moderate capacity distribution



(a) Outdegree 5 (b) Outdegree 10 (c) Outdegree 20

Figure 2: Swaplinks #Nodes vs Selection Frequency for each degree: high churn and moderate capacity distribution

values of churn, and, similarly under two different distributions of node capacities (Section 6.1). Next, we subject both to more demanding churn scenarios: one where network size doubles in the space of 10 seconds, and one where network size halves instantaneously (Section 6.2). We give results of a 250-node experiment over planetlab in Section 6.3. Finally, in Section 6.4 we describe how we can use "smart-pinging" to reduce the heart-beat load incurred by high degree nodes.

In all of these experiments, we evaluate Swaplinks as both a heterogeneous graph construction mechanism (e.g., how well node degrees match desired degrees) and as a heterogeneous peer selection mechanism (e.g. how close the selection probabilities are to the desired values). We evaluate KRB on the other hand as solely a heterogeneous peer selection mechanism.

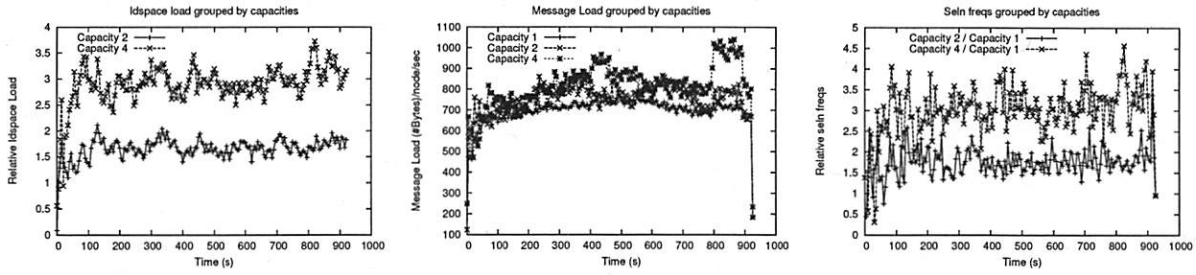
6.1 Evaluation under representative churn scenarios

We use two separate churn scenarios: a "high-churn" scenario in which the median session time is 2 minutes , and a "low-churn" scenario in which the median session time is 30 minutes. These session time values have been taken from previous studies [34, 35, 36]. We similarly use two capacity distributions: (i) The first capacity distribution is a 'moderate' 5:10:20 distribution, with 80% of Swaplinks nodes having outdegree 5, 10% having outdegree 10, and 10% with outdegree 20. We realize the same (relative) capacity split in KRB by having 80% of the nodes have a capacity of 1, 10% have a capacity of 2, and 10% of the nodes have a capacity of 4. (ii) The second capacity distribution is an 'extreme' 3:60:150 distribution, with 98% of the nodes with outdegree 3, 1% of the nodes with outdegree 60, and 1% of the nodes with outdegree 150. Again, we similarly realize the same relative capacity distribution in KRB as well. We restrict the number of high-capacity nodes in the extreme capacity distribution to the relatively small proportion of 1% for the following reason: We run most of our experiments on networks of size 1000. With an increase in the number of high-capacity nodes, it gets more likely that there is a completely connected 'core' made of the high-capacity nodes, and all other nodes directly connected to the core nodes. The fact that this behavior is not retained when the network grows to a larger size (where the network maintains the same capacity distribution) makes such networks not representative of general P2P settings.

We use node session times that are independent of capacities, and follow the Pareto distribution. Networks start from scratch (zero nodes), and total experiment times are typically set to more than 5 times the median

	High Churn						Low Churn					
Target Outdeg	5	10	20	3	60	150	5	10	20	3	60	150
Avg Load(B/s)	150.26	297.87	581.49	98.16	1621.23	3449.41	124.33	247.34	491.49	79.62	1275.89	2903.17
Relative Load	1	1.98	3.86	1	16.43	35.05	1	1.98	3.95	1	15.92	36.20
Avg Totaldeg	9.68	19.41	38.23	5.80	111.77	255.25	9.98	19.93	39.95	5.98	119.98	298.18
Relative Selns	1	2.01	3.95	1	19.78	44.43	1	2.00	3.99	1	20.10	50.16
Seln p-values	0.815	0.862	0.977	0.757	0.579	0.819	0.292	0.784	0.583	0.224	0.957	NaN

Table 1: Swaplinks results for moderate and extreme capacity distributions under high and low churn.



(a) Relative Id Space Distribution

(b) Message Loads

(c) Relative Selection Frequencies

Figure 3: KRB under high churn and moderate capacity distribution

node session times. We ran tests where node session times were dependent on capacities (i.e., where high capacity nodes are likely to stay in the system longer) and where session times were Poisson distributed, and we found the results to be similar to those we present here.

Unless otherwise mentioned, we run ongoing background peer selections, where the 80 longest living nodes perform a random selection every 250 ms for the duration of their lifetime. We call such selections 'periodic' selections. We use the periodic selections to evaluate whether, for instance, the degree 20 nodes receive, in aggregate, twice as many selections as do degree 10 nodes, over the course of the experiment. We also have two other nodes perform a 'burst' of 10,000 selections with a gap of 10 ms between successive selections. We use the short-term burst to obtain a set of selection measurements with relatively little churn. This allows us to more accurately compare the measured distribution of selections among a group of same-capacity nodes with the ideal distribution. This is because each node present in the network during the burst receives a statistically large number of (measured or ideal) selections. The burst selections are performed just before the end of each experiment.

We measure message loads in both Swaplinks and KRB by counting only the bytes in the message payloads; we do not consider TCP/IP or UDP header overheads.

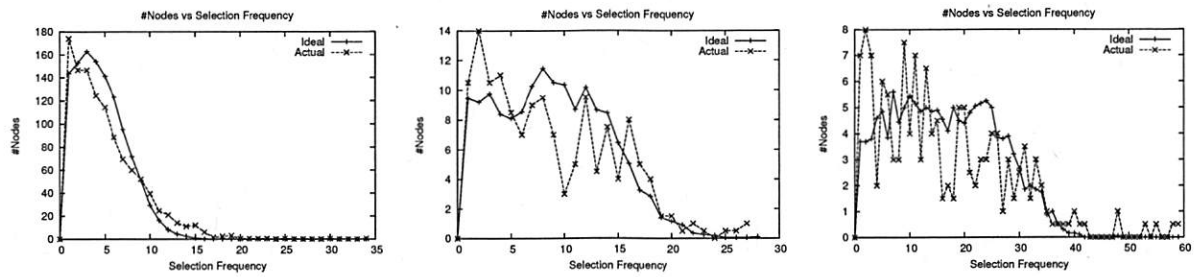
6.1.1 Swaplinks Results

Figures 1 and 2 show the results of the high-churn, moderate capacity distribution experiment for Swaplinks. The node degrees closely track the desired values (Figure 1(a)), while the selections and message loads are split among the different nodes in proportion to their capacities: for example, nodes with outdegree 10 receive twice as many selections, on an average, as the nodes with outdegree 5. Both periodic and burst selections are counted to compute the curves in Figure 1(c).

Figure 2 shows the selection frequencies that result from the burst selections. The figure has one plot for each of the three different capacity classes, where a capacity class is just a set of nodes with the same capacity. The "actual" curve represents the Swaplinks selections. The "ideal" curve represents the ideal distribution of the particular class' 'fair share' of the total number of successful selections; the intersection of each node's lifetime with the time-span of the burst selections is taken into account in computing this distribution. These values don't include failed selections, which occur with churn because nodes take about 10 seconds to detect that a neighbor is down. Thus, the higher the churn-rate is, the greater the probability is that a selection walk fails by being forwarded to a now-dead neighbor at some hop. High churn has about 40%-45% failed selection walks, while low churn has about 2% failed walks.

As can be seen from the plots in figure 2, Swaplinks' actual selection frequency distribution closely tracks the ideal curve for each of the different capacities. This, coupled with the fact that Swaplinks also realizes capacity-wise selection distribution (Figure 1(c)), demonstrates that the selection mechanism realizes the desired distribution.

Table 1 gives a summary of results from all of the Swaplinks experiments in this section by averaging each value over the second half of the experiment time. The duration of high-churn experiments here is around 930



(a) Capacity 1 (b) Capacity 2 (c) Capacity 4

Figure 4: KRB #Nodes vs Selection Frequency for burst selections: high churn and moderate capacity distribution

	High Churn						Low Churn					
Target Split	1	2	4	1	20	50	1	2	4	1	20	50
Idspace Split:	1	1.68	2.99	1	6.14	5.89	1	1.95	3.98	1	23.59	37.41
Msg Load(B/s)	711.30	765.04	853.78	736.63	922.48	923.40	261.73	290.59	318.45	297.01	474.06	413.49
Relative Selns	1	1.78	3.13	1	5.74	5.29	1	1.98	4.02	1	23.23	34.44
Seln p-values	0.000	0.231	0.054	0.000	0.000	0.002	0.000	0.645	0.774	0.000	NaN	0.001

Table 2: KRB results for moderate and extreme capacity distributions under high and low churn.

seconds, whereas the duration of the low-churn experiments is around 14,000 seconds. Each row in the table corresponding to a 'relative' value shows the corresponding value for the capacity class as a ratio over the equivalent value in the lowest capacity class in the experiment. Both periodic selections and burst selections are taken into account in computing the 'Relative-Selns' row. The last row plots the χ^2- test p-values of the selection frequency distribution (for burst selections): this is an indicator of how well the actual selection frequencies of nodes within each capacity class match the ideal selection frequencies. Larger values indicate a closer match; p-values greater than 0.05 are generally believed to indicate a good match of the observed distribution with the expected distribution.

As can be seen from the table, with the one exception of the high-churn extreme-capacity case, node degrees and selection frequencies closely track the desired values. The valid p-values are all comfortably greater than 0.05, indicating good selection distribution.[3]

In the high-churn extreme capacity case, high degree nodes have an average total degree that is less than the respective ideal values: High degree nodes need some time to reach their full degrees upon entering the system, because they have at most 10 neighbor discovery walks outstanding at any time. This effect is more prominent during high-churn, where new nodes enter more frequently. The values for the relative selection frequencies suffer because of the imperfect degree distribution, but they nevertheless are still reasonably close to the target ratios.

The message load ratios in the extreme capacity distribution deviates from the ideal 3:60:150; this is because some of the high-degree neighbors have duplicate links between them, resulting in a reduction of the heart-beat load incurred. This is an artifact of the fact that the total degree of the highest capacity nodes here is non-negligible in comparison to the total number of links in the system, and we expect the number of duplicate links to decrease and the load-ratios to get closer to the 3:60:150 proportion in larger networks.

Looking at the message loads from an alternate perspective, the absolute values of the message loads for the outdegree 60 and outdegree 150 nodes seem relatively high. The bulk of this load is caused by neighbor heart-beats. In Section 6.4 we describe how we can reduce this load by doing heart-beats in a more sophisticated fashion.

We ran similar experiments for 5000 nodes Swaplinks graphs over a 20 CPU cluster on the Emulab testbed, and found the results to be broadly similar, demonstrating that Swaplinks retains its properties in larger networks as well.

6.1.2 KRB Results

We make a few changes to the parameters used by the default Bamboo source distribution to get KRB to approach the desired relative capacity-wise ID space distributions. For the high-churn results shown in this section, we set ping and leaf-set-alarm periods in Bamboo to 1 second. We set the near and far routing table alarm periods to 2.5 and 5 times the leaf-set-alarm respectively (these ratios are based on the values in Bamboo's code distribution). We use a period of 5 seconds between successive KRB load messages sent to random locations in the network (we denote this period the *KRB period*). For the low-churn results, we set the ping period to 2 seconds, the leaf-set alarm period to 3 seconds and the KRB period to

Target Out-deg	Avg Load(B/s)	Relative Load	Avg Total deg	Relative Selns	Seln p values
	Flash Crowd				
5	146.88	1	9.82	1	0.936
10	291.94	1.98	19.73	2.06	0.935
20	578.49	3.93	39.18	4.02	0.722
3	91.29	1	5.9	1	0.751
60	1553.79	17.01	114.61	19.66	0.873
150	3320.49	36.34	269.54	46.94	0.567
	Mass Departures				
5	179.52	1	9.54	1	0.457
10	351.01	1.95	19.11	2.04	0.912
20	681.62	3.79	37.59	3.96	0.988
3	121.57	1	5.73	1	0.338
60	1886.01	15.5	102.19	17.87	0.418
150	4343.61	35.66	251.1	45.84	NaN

Table 3: Swaplinks performance with flash-crowds and mass departures under high churn.

10 seconds. Bamboo has a time-out value between when a node suspects a neighbor to be down (as a result of failure of message delivery) to when it actually decides it's down (as a result of lack of response to pings sent to the neighbor). We reduce this timeout value to 1 second from the earlier value of 60 seconds. We use a leaf-set size of 4 and a KRB-threshold value (see Section 5) of 0.2 in all KRB simulations. We restricted KRB low-churn simulations to a shorter duration of 1800 seconds; longer simulations took unreasonably longer (wall-clock) times to complete.

Figures 3 and 4, and Table 2 show the results for KRB. The results show that KRB is not successful in maintaining the relative ID-spaces at the desired levels under high churns – it is only able to achieve around a 1:1.65:3 relative division in the ID spaces in the moderate capacity distribution, while its response to the extreme capacity distribution under high churn is worse. KRB is able to achieve the desired relative ID-space distribution in the low-churn moderate-capacity case, but again fails to fully achieve the desired ID-space distribution in the extreme-capacity low-churn scenario. KRB also fails to consistently achieve the desired selection distribution within each capacity class, as seen by the burst selection p-values computed for the selection frequencies. The p-values for the lowest capacity class in the moderate capacity distribution is 0 in both the high and low churn scenarios. This is mainly because the actual selection distributions here have quite a few outliers – nodes with an actual selection frequency close to or greater than the maximum selection frequency (for any node) predicted by the ideal curve. KRB's selection frequency curves within capacity classes 2 and 4 do match the ideal curve closely enough that they succeed the p-value test, but during high-churn, nodes in the higher capacity classes are still less likely to get selected than they should ideally be.

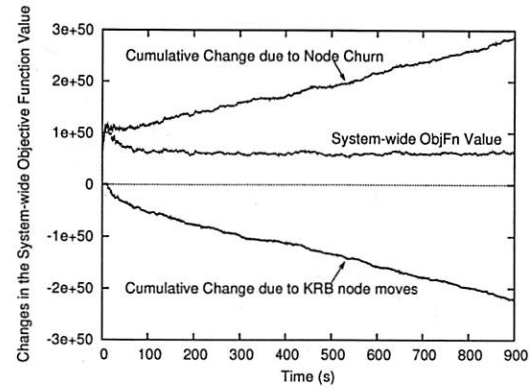
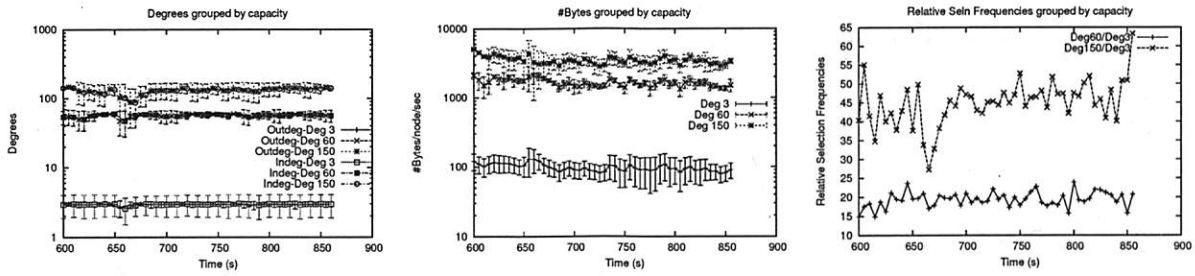


Figure 5: Change in the universal objective function as a result of KRB node moves and node churn in a high-churn, moderate-capacity simulation.

Figure 5 shows why KRB underperforms under high churn: The system-wide objective function (Equation 3, Section 5) settles to a more or less stable positive value in the presence of the steady churn. KRB's attempts to improve the objective function value below this stable value using node movements are exactly counterbalanced by the effects of node churn, indicating that this is the best KRB can do under this high churn. Increasing the frequency of KRB node movements here does not lead to an improvement in performance, as becomes clear next.

We evaluated the relative ID-space distribution realized by KRB under high churn and moderate capacities for various values of the KRB parameters (ping, alarm, KRB periods, KRB-threshold), and we found that the combination of the parameters we present here results in the best ID-space distribution. In general, we found that more frequent pings and alarm messages of Bamboo resulted in better results (as can be expected), while there generally was an 'optimal' KRB message frequency and an optimal value for the KRB threshold given the frequencies used for the other messages. Setting the KRB message frequency to higher values resulted in an increase of the number of incorrect KRB moves, where nodes switched positions based on an incorrectly perceived local state, thereby worsening the ID-space distribution. Among the combinations of parameters we tested, the worst performing set yielded about 50% less accurate selection than the setting we use. This experience indicates that it is harder with KRB to decide on the exact set of various parameters to use in a general setting.[4]

KRB achieves a higher average message load (across all nodes) than does Swaplinks: this is mainly a result of the increased message rates we used to improve KRB's capacity-based ID space distribution. We however do not think that the message load values are high enough to be a concern here.



(a) Degree Distribution (b) Message Loads (c) Relative Selection Frequencies

Figure 6: Swaplinks: Flash-crowd with high churn and extreme capacity distribution

	Flash-crowd			Mass Departures		
Target Split	1	2	4	1	2	4
Idspace Loads	1	1.52	2.41	1	1.19	1.68
Relative Selns	1	1.55	2.47	1	1.14	1.75
Seln p-values	0.00	0.00	0.01	0.00	0.48	0.00

Table 4: KRB Results for flash-crowds and mass departures for moderate capacity distributions and high churn

6.2 Extreme churn

We now look at the reaction of Swaplinks and KRB to more extreme churn events, the first where a flash-crowd leads to the network size doubling from 1000 nodes to 2000 nodes in the span of 10 seconds, and the second where a half of the network dies instantaneously.

Figure 6 shows the results of the Swaplinks flash-crowd scenario under a 3:60:150 degree distribution under high-churn (a median node session time of 2 minutes). The flash-crowd appears in the period 650-660 seconds after the system is started, and two sets of burst-selections are performed starting at 723 seconds and spanning 100 seconds. Table 3 summarizes the flash-crowd results over the last 175 seconds of the experiment for both the moderate and extreme capacity distributions.

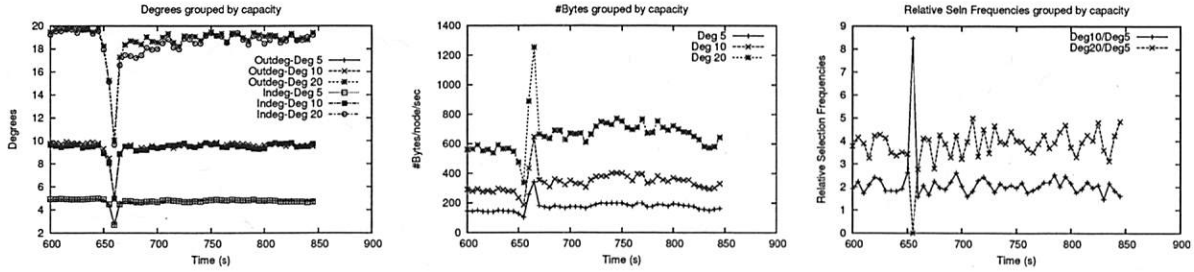
Figure 6 shows that while there is a temporary deterioration in all the metrics of interest for a short duration of time immediately after the entry of the flash-crowd, the system quickly recovers to re-establish desired behavior. The Swaplinks graph in fact generally benefits from nodes entering the system, since this pushes the average degree distribution across the graph towards the ideal value; a comparison of Table 3 with Table 1 shows that the average values for the degree and relative selection frequencies in fact improve as a result of the arrival of the flash-crowd!

Figure 7 and Table 3 show results of the Swaplinks mass departure experiments. The mass departures occur at 649 seconds after system start, and burst selections are performed at 719 seconds after system start. From figure 7 (for high churn and moderate capacity distributions), we observe that the network suffers for a short duration of time immediately after the huge perturbation, but things start to improve thereafter. The message loads and the selection frequencies recover to re-approach the desired 1:2:4 split of message loads and selection frequencies. The extreme capacity results from Table 3 also look encouraging: the degrees and the relative selection frequencies are similar to the high (stable) churn, extreme-capacity results shown earlier in Table 1. Overall, these experiments demonstrate that Swaplinks is robust to various kinds of network churn under widely different capacity distributions, and that it manages to retain its fine-grained sensitivity to the desired heterogeneity under these conditions.

Table 4 summarizes KRB results from the last 175 seconds of the flash-crowd and the mass departure simulations for only the moderate capacity distribution under high churn. The flash-crowds and mass departures occur at the same times as those reported in the Swaplinks experiments. The results indicate that the KRB performance suffers significantly as a result of the extreme churn induced. The relative ID-spaces and selection frequencies differ markedly from the target values, resulting in a failure to realize the desired selection distribution. We noticed that while KRB had started to recover from the flash-crowd to approach its stable ID-space distribution towards the end of the simulation, in the mass departure simulation its stable ID-space distribution deteriorated after the mass departures, leading to worse relative selection values at the end of the simulation. [5] Since we have already seen that KRB fails to adapt to the extreme-capacity setting under high churn, we do not subject it to the more demanding circumstances of both extreme churn (mass departures and flash crowds) and extreme heterogeneity.

6.3 Evaluation over PlanetLab

We evaluated Swaplinks over PlanetLab by deploying a 250-node network over 50 PlanetLab hosts distributed across the world. We scaled down the number of selec-



(a) Degree Distribution (b) Message Loads (c) Relative Selection Frequencies

Figure 7: Swaplinks: Mass departures with high churn and moderate capacity distribution

	High Churn			Low Churn		
Target Outdeg	5	10	20	5	10	20
Avg Load(B/s)	210.88	418.36	794.16	196.81	384.10	745.75
Load split	1	1.97	3.76	1	1.95	3.76
TotalDeg	9.54	19.15	37.46	10.04	19.79	39.48
Relative Selns	1	2.07	3.99	1	2.01	3.94
Seln p-values	0.000	0.000	0.001	0.000	0.023	0.002

Table 5: PlanetLab results with moderate capacity distribution

tions performed in the burst mode here to about 2500.

Figure 8 shows the variation of average node degrees, message loads and the relative selection frequencies with time in a high-churn moderate capacity experiment , and Table 5 summarizes both the high-churn and low-churn experiments. While the node-degree curve in the high churn case is not completely stable, due to the high churn, all the values nevertheless adhere reasonably closely to the desired 5:10:20 ratio. But there is a gap between the ideal distribution of #Nodes vs Selection Frequencies and the actual distribution here, leading to poor p-values for the selection distribution. We observed that a few of the planetlab nodes hosting our experiments appeared to freeze occasionally, causing the Swaplinks instances hosted on these nodes to be eventually excluded from the neighbor-sets of other Swaplinks instances. This also means that such nodes would not be selected by any subsequently launched random selection walk, thus causing the discrepancy between the actual and observed selection distributions.

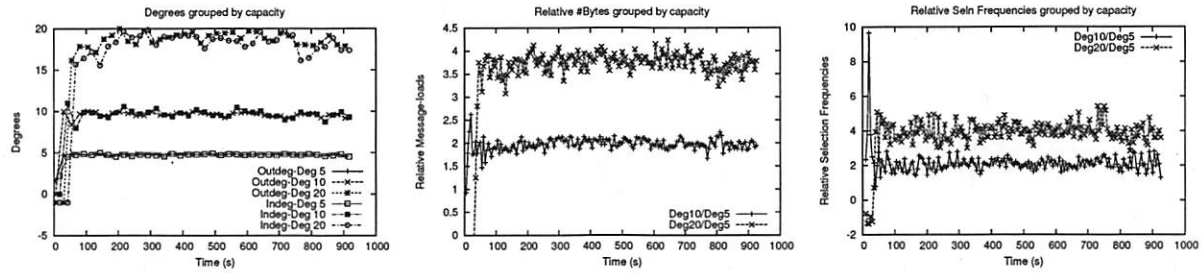
We do not show results for the extreme capacity distribution here: the fact that each high capacity class constitutes just 1% of the total node population means that there would be too few high capacity nodes in a 250-node experiment to draw reliable conclusions.

6.4 Smart-Pinging

The bulk of the message load seen by Swaplinks nodes is from the heart-beat messages used to determine when a neighbor is down. We would like to minimize this load, in part because in extreme heterogeneity situations some nodes have many neighbors, but in part because a given application might result in a computer belonging to many P2P networks, and therefore have many neighbors. Our basic approach to minimizing heart-beats is as follows: Rather than have every neighbor determine for itself whether a node A is down, one neighbor (at a time) determines if a node A is down. If a neighbor determines that node A is down, it informs the other neighbors of node A, using a flood, that node A is down.

Specifically, the *smart-pinging* scheme we designed works as follows: Node A tells each of its neighbors about some random set of its other neighbors, such that each neighbor is known by at least some small number of other neighbors. Node A sends each neighbor in turn a small series of (say five) heart-beat messages, each spread two seconds apart. For example, node A sends neighbor 1 five heartbeats, neighbor 2 five heartbeats, and so on. Each neighbor knows when to expect its series of heartbeats, based on timing information conveyed during the previous series of heartbeats. If a neighbor misses all of its heartbeats, it informs all the neighbors of A it knows of that node A is down. These neighbors in turn inform the neighbors they know, and the ensuing flood of packets quickly informs all neighbors that node A is down.[6]

Smart-pinging reduces the amount of bandwidth consumed under no churn, at the cost of a burst of messages that occurs when there is churn, and the possibility of incorrect notifications of node departure. While we need to explore these trade-offs in greater detail, we have currently implemented a preliminary version of smart-



(a) Degree Distribution (b) Message Loads (c) Relative Selection Frequencies

Figure 8: PlanetLab 250 nodes with high churn and moderate capacity distribution

Target Outdeg	5	10	20
Avg Load (B/s)	38.23	57.97	89.21
Relative Load	1	1.50	2.30
Avg Totaldeg	9.99	19.90	40.01
Relative Selns	1	2.06	4.08
Seln p-values	0.831	0.904	0.877

Table 6: Smart Pinging: moderate capacity, low churn

pinging, and observe that it does indeed result in a saving on message load under low-churn scenarios. In the current implementation, if node A has d out-neighbors, A has each of its neighbors know of $2\,log_2(d)$ of its (A's) neighbors. Table 6 summarizes the results over the second half of the duration of the experiment. This experiment was run with just 8 periodic selectors (instead of 80 as in the previous cases), to isolate the heart-beat load.

7 Conclusions

Node heterogeneity, where different nodes have different capacities, is an important issue in current peer-to-peer systems. In this paper, we provide the implementation and performance evaluation of the Swaplinks heterogeneous graph construction and peer selection mechanism. We also compare its heterogeneous selection properties with that of *KRB*, a structured P2P approach derived by adapting the Karger-Ruhl load-balancing scheme to node ID spaces in the Bamboo DHT.

We find that while Swaplinks generally gives good performance along all metrics of interest, KRB finds it hard, under relatively high churn rates, to maintain the desired selection probabilities even for moderate distributions in desired selection probabilities. Also, with KRB, it is non-trivial to zero in on a good set of tuning parameters to use in a general setting. Overall, we find that Swaplinks outperforms KRB in performing heterogeneity-sensitive random peer selection.

In terms of enhancements to Swaplinks, we need to experiment further with smart-pinging, for instance to insure that it doesn't suffer from false negatives. In addition, we note that Swaplinks discovers truly random neighbors. Some P2P applications, however, would like to also discover neighbors that are nearby in terms of latency. While a P2P application is free to do that on its own (i.e. by using Swaplinks to discover random peers, and then measuring latency to them), we believe that it would beneficial to explore efficient ways to do this, and add the capability to the Swaplinks toolkit.

A limitation of Swaplinks is that it has no defense against misbehaving nodes. For instance, if a node wished to obtain a huge number of neighbors (for instance to DoS a file-sharing application), Swaplinks has no mechanism to prevent this. While we are interested in exploring such mechanisms, Swaplinks is currently only appropriate for use with trusted P2P software.

We are currently exercising Swaplinks by using it as a basis for a number of P2P applications: The Swaplinks toolkit is being used as the basis for the Chunkyspread P2P multicast system [7]. In addition, the Swaplinks algorithm is being used in building a toolkit for NAT traversal in P2P applications, and a P2P file backup system. We invite researchers to use our Swaplinks toolkit in their unstructured P2P applications [16].

8 Acknowledgments

We would like to thank Grant Goodale and Victoria Krafft for help in adapting the Bamboo simulator to KRB.

Notes

[1]To be fair, neither of these schemes expressly aim to maintain node degrees perfectly proportional to capacities.

[2]For the results shown in this paper, we do not utilize the TCP socket close signal as an indicator of neighbor departure, so as to have a fair comparison with KRB, since Bamboo uses UDP

[3]The single "NaN" entry indicates that there were too few (<5) nodes of the particular capacity during the time when the burst selections were performed for a meaningful p-value to be computed

[4]In the search for the best combination of KRB parameters, we did not try out sub-second values for the different parameters: We could conceivably use sub-second values, and achieve better results, but we did not consider this option due to the enormous amount of load it places on the network.

[5]The single positive p-value result here seems to be a lucky one for the nodes in the second capacity class – the smallest capacity nodes get more of the selections than their fair share while the largest get fewer, leaving the capacity 2 nodes with the number of selections closest to its fair share (while still less than it)

[6] Structella [37] uses a similar mechanism to reduce heart-beat loads in maintaining leaf-sets, but their mechanism is not applicable in maintaining any arbitrary set of neighbors.

References

[1] Paul Francis. Yoid: Extending the internet multicast architecture. In *Unrefereed report*, 2000.

[2] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. ACM SOSP*, 2003.

[3] Vinay Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, and Alexander E. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *Proc. IPTPS*, 2005.

[4] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *Proc. ACM SIGCOMM*, 2003.

[5] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *Proc. ACM SIGCOMM*, 2004.

[6] Li wei Lehman and Steven Lerman. Palm: Predicting internet network distances using peer-to-peer measurements. In *Technical Report, MIT*, 2004.

[7] Vidhyashankar Venkataraman and Paul Francis. Chunkyspread: Heterogeneous unstructured tree-based peer to peer multicast. In *Proc. ICNP*, 2006.

[8] Yang hua Chu, Aditya Ganjam, T. S. Eugene Ng, Sanjay G. Rao, Kunwadee Sripanidkulchai, Jibin Zhan, and Hui Zhang. Early deployment experience with an overlay based internet broadcasting system. In *Proc. USENIX Annual Technical Conference*, 2004.

[9] A. Bharambe, S. Rao, V. Padmanabhan, S. Seshan, and H. Zhang. The impact of heterogeneous bandwidth constraints on dht-based multicast protocols. In *Proc. IPTPS*, 2005.

[10] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, , and A. Singh. Splitstream: High-bandwidth content distribution in cooperative environments. In *Proc. ACM SOSP*, 2003.

[11] Vivek Vishnumurthy and Paul Francis. On heterogeneous overlay construction and random node selection in unstructured p2p networks. In *Proc. IEEE Infocom*, 2006.

[12] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proc. IPTPS*, 2004.

[13] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *Proc. USENIX Annual Technical Conference*, 2004.

[14] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM*, 2001.

[15] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, 2001.

[16] http://www.cs.cornell.edu/%7evivi/research/swaplinks.html.

[17] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM 2001*, 2001.

[18] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. ACM STOC*, 1997.

[19] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP*, 2001.

[20] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard M. Karp, and Ion Stoica. Load balancing in structured p2p systems. In *IPTPS*, 2003.

[21] P. Brighten Godfrey and Ion Stoica. Distributed construction of random expander networks. In *Proc. IEEE Infocom*, 2005.

[22] Jonathan Ledlie and Margo Seltzer. Distributed, secure load balancing with skew, heterogeneity, and churn. In *Proc. IEEE Infocom*, 2005.

[23] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proc. NSDI*, 2005.

[24] Miguel Castro, Manuel Costa, and Antony Rowstron. Debunking some myths about structured and unstructured overlays. In *Proc. NSDI*, 2005.

[25] David R. Karger and Matthias Ruhl. New algorithms for load balancing in peer-to-peer systems. In *IRIS Student Workshop (ISW '03)*, 2003.

[26] Roie Melamed and Idit Keidar. Araneola: A scalable reliable multicast system for dynamic environments. In *Proc. NCA 2004*, 2004.

[27] C. Law and K.-Y. Siu. Distributed construction of random expander networks. In *Proc. IEEE Infocom*, 2003.

[28] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulie. Scamp: peer-to-peer lightweight membership service for large-scale group communication. In *Proc. 3rd Intnl. Wshop Networked Group Communication (NGC '01)*, pages 44–55, 2001.

[29] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using random subsets to build scalable network services. In *Proc. USITS*, 2003.

[30] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equations of state calculations by fast computing machines. In *Journal of Chemical Physics*, 1953.

[31] I Csiszár. Information theoretic methods in probability and statistics. In *IEEE Information Theory Society Newsletter 48*, 1998.

[32] Saikat Guha and Paul Francis. Towards a Secure Internet Architecture Through Signaling. Technical Report cul.cis/TR2006-2037, Cornell University, 2006.

[33] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *Proc. IEEE Infocom*, 1996.

[34] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. In *Second Annual ACM Internet Measurement Workshop*, 2002.

[35] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. MMCN*, 2002.

[36] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. ACM SOSP*, 2003.

[37] Miguel Castro, Manuel Costa, and Antony Rowstron. Should we build gnutella on a structured overlay? In *Proc. HotNets-II*, 2003.

Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems

Oren Laadan
Department of Computer Science
Columbia University

Jason Nieh
Department of Computer Science
Columbia University

Abstract

The ability to checkpoint a running application and restart it later can provide many useful benefits including fault recovery, advanced resources sharing, dynamic load balancing and improved service availability. However, applications often involve multiple processes which have dependencies through the operating system. We present a transparent mechanism for commodity operating systems that can checkpoint multiple processes in a consistent state so that they can be restarted correctly at a later time. We introduce an efficient algorithm for recording process relationships and correctly saving and restoring shared state in a manner that leverages existing operating system kernel functionality. We have implemented our system as a loadable kernel module and user-space utilities in Linux. We demonstrate its ability on real-world applications to provide transparent checkpoint-restart functionality without modifying, recompiling, or relinking applications, libraries, or the operating system kernel. Our results show checkpoint and restart times 3 to 55 times faster than OpenVZ and 5 to 1100 times faster than Xen.

1 Introduction

Application checkpoint-restart is the ability to save the state of a running application to secondary storage so that it can later resume its execution from the time at which it was checkpointed. Checkpoint-restart can provide many potential benefits, including fault recovery by rolling back an application to a previous checkpoint, better application response time by restarting applications from checkpoints instead of from scratch, and improved system utilization by stopping long running computationally intensive jobs during execution and restarting them when load decreases. An application can be migrated by checkpointing it on one machine and restarting it on another providing further benefits, including fault resilience by migrating applications off of faulty hosts, dynamic load balancing by migrating applications to less loaded hosts, and improved service availability and administration by migrating applications before host maintenance so that applications can continue to run with minimal downtime.

Many important applications consist of multiple cooperating processes. To checkpoint-restart these applications, not only must application state associated with each process be saved and restored, but the state saved and restored must be globally consistent and preserve process dependencies. Operating system process state including shared resources and various identifiers that define process relationships such as group and session identifiers must be saved and restored correctly. Furthermore, for checkpoint-restart to be useful in practice, it is crucial that it transparently support the large existing installed base of applications running on commodity operating systems.

Zap [16] is a system that provides transparent checkpoint-restart of unmodified applications that may be composed of multiple processes on commodity operating systems. The key idea is to introduce a thin virtualization layer on top of the operating system that encapsulates a group of processes in a virtualized execution environment and decouples them from the operating system. This layer presents a host-independent virtualized view of the system so that applications can make full use of operating system services and still be checkpointed then restarted at a later time on the same machine or a different one. While previous work [16] presents key aspects of Zap's design, it did not describe a number of important engineering issues in building a robust checkpoint-restart system. In particular, a key issue that was not discussed is how to transparently checkpoint multiple processes such that they can be restarted in a globally consistent state.

Building on Zap, we address this consistency issue and discuss in detail for the first time key design issues in building a transparent checkpoint-restart mechanism for multiple processes on commodity operating systems. We combine a kernel-level checkpoint mechanism with a hybrid user-level and kernel-level restart mechanism to leverage existing operating system interfaces and functionality as much as possible for checkpoint-restart. We introduce a novel algorithm for accounting for process relationships that correctly saves and restores all process state in a globally consistent manner. This algorithm is crucial for enabling transparent checkpoint-restart of interactive graphical applications and correct job control. We introduce an efficient algorithm for identifying and accounting for shared resources and correctly saving and restoring such shared state across cooperating processes. To

reduce application downtime during checkpoints, we also provide a copy-on-write mechanism that captures a consistent checkpoint state and correctly handles shared resources across multiple processes.

We have implemented a checkpoint-restart prototype as a set of user-space utilities and a loadable kernel module in Linux that operates without modifying, recompiling, or relinking applications, libraries, or the operating system kernel. Our measurements on a wide range of real-world server and desktop Linux applications demonstrate that our prototype can provide fast checkpoint and restart times with application downtimes of less than a few tens of milliseconds. Comparing against commercial products, we show up to 12 times faster checkpoint times and 55 times faster restart times than OpenVZ [15], another operating system virtualization approach. We also show up to 550 times faster checkpoint times and 1100 faster restart times than Xen [5], a hardware virtualization approach.

This paper is organized as follows. Section 2 discusses related work. Section 3 provides background on the Zap virtualization architecture. Section 4 provides an overview of our checkpoint-restart architecture. Section 5 discusses how processes are quiesced to provide a globally consistent checkpoint across multiple processes. Section 6 presents the algorithm for saving and restoring the set of process relationships associated with a checkpoint. Section 7 describes how shared state among processes is accounted for during checkpoint and restart. Section 8 presents experimental results on server and desktop applications. Finally, we present some concluding remarks.

2 Related Work

Many application checkpoint-restart mechanisms have been proposed [17, 22, 23]. Application-level mechanisms are directly incorporated into the applications, often with the help of languages, libraries, and preprocessors [2, 7]. These approaches are generally the most efficient, but they are non-transparent, place a major burden on the application developer, may require the use of nonstandard programming languages [8], and cannot be used for unmodified or binary-only applications.

Library checkpoint-restart mechanisms [18, 28] reduce the burden on the application developer by only requiring that applications be compiled or relinked against special libraries. However, such approaches do not capture important parts of the system state, such as interprocess communication and process dependencies through the operating system. As a result, these approaches are limited to being used with applications that severely restrict their use of operating system services.

Operating system checkpoint-restart mechanisms utilize kernel-level support to provide greater application transparency. They do not require changes to the application source code nor relinking of the application object code, but they do typically require new operating systems [9, 10] or invasive kernel modifications [15, 21, 24, 26] in commodity operating systems. None of these approaches checkpoints multiple processes consistently on unmodified commodity operating systems. Our work builds on Zap [16] to provide transparent checkpoint-restart functionality by leveraging loadable kernel module technology and operating system virtualization. The operating system virtualization approach introduced in Zap is also becoming popular for providing isolation containers for groups of processes while allowing scalable use of system resources [15, 20, 12].

Hardware virtualization approaches such as Xen [5] and VMware [29] use virtual machine monitors (VMMs) [19] that can enable an entire operating system environment, both operating system and applications, to be checkpointed and restarted. VMMs can support transparent checkpoint-restart of both Linux and Windows operating systems. However, because they operate on entire operating system instances, they cannot provide finer-granularity checkpoint-restart of individual applications decoupled from operating system instances, resulting in higher checkpoint-restart overheads and differences in how these mechanisms can be applied.

3 Virtualization Architecture

To enable checkpoint-restart, we leverage Zap's operating system virtualization. Zap introduces a thin virtualization layer between applications and the operating system to decouple applications from the underlying host. The virtualization layer provides a pod (PrOcess Domain) virtual machine abstraction which encapsulates a set of processes in a self-contained unit that can be isolated from the system, checkpointed to secondary storage, and transparently restarted later. This is made possible because each pod has its own virtual private namespace, which provides the only means for processes to access the underlying operating system. To guarantee correct operation of unmodified applications, the pod namespace provides a traditional environment with unchanged application interfaces and access to operating system services and resources.

Operating system resource identifiers, such as process IDs, must remain constant throughout the life of a process to ensure its correct operation. However, when a process is checkpointed and later restarted, possibly on a different operating system instance, there is no guarantee that the system will provide the same identifiers to the restarted process; those identifiers may in fact be in use by other processes in the system. The pod namespace addresses these issues by providing consistent, virtual resource names. Names within a pod are trivially assigned in a unique manner in the same way that traditional oper-

ating systems assign names, but such names are localized to the pod. These virtual private names are not changed when a pod is restarted to ensure correct operation. Instead, pod virtual resources are transparently remapped to real operating system resources.

In addition to providing a private virtual namespace for processes in a pod, our virtualization approach provides three key properties so that it can be used as a platform for checkpoint-restart. First, it provides mechanisms to synchronize the virtualization of state across multiple processes consistently with the occurrence of a checkpoint, and upon restart. Second, it allows the system to select predetermined virtual identifiers upon the allocation of resources when restarting a set of processes so that those processes can reclaim the same set of virtual resources they had used prior to the checkpoint. Third, it provides virtualization interfaces that can be used by checkpoint and restart mechanisms to translate between virtual identifiers and real operating system resource identifiers. During normal process execution, translating between virtual and real identifiers is private to the virtualization layer. However, during checkpoint-restart, the checkpoint and restart functions may also need to request such translations to match virtual and real namespaces.

4 Checkpoint-Restart Architecture

We combine pod virtualization with a mechanism for checkpointing and restarting multiple cooperating processes in a pod. For simplicity, we describe the checkpoint-restart mechanism assuming a shared storage infrastructure across participating machines. In this case, filesystem state is not generally saved and restored as part of the pod checkpoint image to reduce checkpoint image size. Available filesystem snapshot functionality [14, 6] can be used to also provide a checkpointed filesystem image. We focus only on checkpointing process state; details on how to checkpoint filesystem, network, and device state are beyond the scope of this paper.

Checkpoint: A checkpoint is performed in the following steps:

1. *Quiesce pod*: To ensure that a globally consistent checkpoint [3] is taken of all the processes in the pod, the processes are quiesced. This forces the processes to transfer from their current state to a controlled standby state to freeze them for the duration of the checkpoint.
2. *Record pod properties*: Record pod configuration information, in particular filesystem configuration information indicating where directories private to the pod are stored on the underlying shared storage infrastructure.
3. *Dump process forest*: Record process inheritance relationships, including parent-child, sibling, and process session relationships.
4. *Record globally shared resources*: Record state associated with shared resources not tied to any specific process, such as System V IPC state, pod's network address, hostname, system time and so forth.
5. *Record process associated state*: Record state associated with individual processes and shared state attached to processes, including process run state, program name, scheduling parameters, credentials, blocking and pending signals, CPU registers, FPU state, `ptrace` state, filesystem namespace, open files, signal handling information, and virtual memory.
6. *Continue pod*: Resume the processes in the pod once the checkpoint state has been recorded to allow the processes to continue executing, or terminate the processes and the pod if the checkpoint is being used to migrate the pod to another machine. (If a filesystem snapshot is desired, it is taken prior to this step.)
7. *Commit data*: Write out buffered recorded data (if any) to storage (or to the network) and optionally force flush of the data to disk.

To reduce application downtime due to the pod being quiesced, we employ a lazy approach in which the checkpoint data is first recorded and buffered in memory. We defer writing it out to storage (or to the network) until after the pod is allowed to continue, thereby avoiding the cost of expensive I/O operations while the pod is quiesced. Since allocation of large memory chunks dynamically can become expensive too, buffers are preallocated before the pod is quiesced, based on an estimate for the required space. The data accumulated in the buffer is eventually committed in step 7 after the pod has been resumed.

We use a standard copy-on-write (COW) mechanism to keep a reference to memory pages instead of recording an explicit copy of each page. This helps to reduce memory pressure and avoids degrading cache performance. It reduces downtime further by deferring the actual memory to memory copy until when the page is either modified by the application or finally committed to storage, whichever occurs first. Using COW ensures that a valid copy of a page at time of checkpoint remains available if the application modifies the page after the pod has resumed operation but before the data has been committed. Pages that belong to shared memory regions cannot be made copy-on-write, and are handled by recording an explicit copy in the checkpoint buffer. Note that we do not use the `fork` system call for creating a COW clone [10, 18, 26] because its semantics require a process to execute the call itself. This cannot be done while the process is in a controlled standby state, making it difficult to ensure global consistency when checkpointing multiple processes.

The contents of files are not normally saved as part of the checkpoint image since they are available on the filesystem. An exception to this rule are open files that

have been unlinked. They need to be saved during checkpoint since they will no longer be accessible on the filesystem. If large unlinked files are involved, saving and restoring them as part of the checkpoint image incurs high overhead since the data needs to be both read then written during both checkpoint and restart. To avoid these data transfer costs, we instead relink the respective inode back to the filesystem. To maintain the illusion that the file is still unlinked, it is placed in a protected area that is unaccessible to processes in the pod. If relinking is not feasible, such as if a FAT filesystem implementation is used that does not support hard links, we cannot relink but instead store the unlinked file contents in a separate file in a protected area. This is still more efficient than including the data as part of the checkpoint image.

Restart: Complementary to the checkpoint, a restart is performed in the following steps:

1. *Restore pod properties*: Create a new pod, read pod properties from the checkpoint image and configure the pod, including restoring its filesystem configuration.
2. *Restore process forest*: Read process forest information, create processes at the roots of the forest, then have root processes recursively create their children.
3. *Restore globally shared resources*: Create globally shared resources, including creating the necessary virtual identifiers for those resources.
4. *Restore process associated state*: Each created process in the forest restores its own state then quiesces itself until all other processes have been restored.
5. *Continue*: Once all processes in the pod are restored, resume them so they can continue execution.

Before describing the checkpoint-restart steps in further detail, we first discuss three key aspects of their overall structure: first, whether the mechanism is implemented at kernel-level or user-level; second, whether it is performed within the context of each process or by an auxiliary process; and finally the ordering of operations to allow streaming of the checkpoint data.

Kernel-level vs user-level: Checkpoint-restart is performed primarily at kernel-level, not at user-level. This provides application transparency and allows applications to make use of the full range of operating system services. The kernel-level functionality is explicitly designed so that it can be implemented as a loadable module without modifying, recompiling, or relinking the operating system kernel. To simplify process creation, we leverage existing operating system services to perform the first phase of the restart algorithm at user-level. The standard process creation system call `fork` is used to reconstruct the process forest.

In context vs auxiliary: Processes are checkpointed from outside of their context and from outside of the pod using a separate auxiliary process, but processes are restarted from inside the pod within the respective context of each process. We use an auxiliary process that runs outside of the pod for two reasons. First, since all processes within the pod are checkpointed, this simplifies the implementation by avoiding the need to special case the auxiliary process from being checkpointed. Second, the auxiliary process needs to make use of unvirtualized operating system functions to perform parts of its operation. Since processes in a pod are isolated from processes outside of the pod when using the standard system call interface [16], the auxiliary process uses a special interface for accessing the processes inside of the pod to perform the checkpoint.

Furthermore, checkpoint is done not within the context of each process for four reasons. First, using an auxiliary process makes it easier to provide a globally consistent checkpoint across multiple processes by simply quiescing all processes then taking the checkpoint; there is no need to run each process to checkpoint itself and attempt to synchronize their checkpoint execution. Second, a set of processes is allowed to be checkpointed at any time and not all of the processes may be runnable. If a process cannot run, for example if it is stopped at a breakpoint as a result of being traced by another process, it cannot perform its own checkpoint. Third, to have checkpoint code run in the process context, the process must invoke this code involuntarily since we do not require process collaboration. While this can be addressed by introducing a new specific signal to the kernel [11] that is served within the kernel, it requires kernel modifications and cannot be implemented by a kernel module. Fourth, it allows for using multiple auxiliary processes concurrently (with simple synchronization) to accelerate the checkpoint operation.

Unlike checkpoint, restart is done within the context of the process that is restarted for two reasons. First, operating within process context allows us to leverage the vast majority of available kernel functionality that can only be invoked from within that context, including almost all system calls. Although checkpoint only requires saving process state, restart is more complicated as it must create the necessary resources and reinstate their desired state. Being able to run in process context and leverage available kernel functionality to perform these operations during restart significantly simplifies the restart mechanism. Second, because the restart mechanism creates a new set of processes that it completely controls on restart, it is simple to cause those processes to run, invoke the restart code, and synchronize their operations as necessary. As a result, the complications with running in process context during checkpoint do not arise during restart.

More specifically, restart is done by starting a supervisor process which creates and configures the pod, then injects itself into the pod. Once it is in the pod, the supervisor forks the processes that constitute the roots of the process forest. The root processes then create their chil-

dren, which recursively create their descendants. Once the process forest has been constructed, all processes switch to operating at kernel-level to complete the restart. The supervisor process first restores globally shared resources, then each process executes concurrently to restore its own process context from the checkpoint. When all processes have been restored, the restart completes and the processes are allowed to resume normal execution.

Data streaming: The steps in the checkpoint-restart algorithm are ordered and designed for streaming to support their execution using a sequential access device. Process state is saved during checkpoint in the order in which it needs to be used during restart. For example, the checkpoint can be directly streamed from one machine to another across the network and then restarted. Using a streaming model provides the ability to pass checkpoint data through filters, resulting in extremely flexible and extensible architecture. Example filters include encryption, signature/validation, compression, and conversion between operating system versions.

5 Quiescing Processes

Quiescing a pod is the first step of the checkpoint, and is also the last step of the restart as a means to synchronize all the restarting processes and ensure they are all completely restored before they resume execution. Quiescing processes at checkpoint time prevents them from modifying system state, and thus prevents inconsistencies from occurring during the checkpoint. Quiescing also puts processes in a known state from which they can easily be restarted. Without quiescing, checkpoints would have to capture potentially arbitrary restart points deep in the kernel, wherever a process might block.

Processes are quiesced by sending them a `SIGSTOP` signal to force them into the stopped state. A process is normally either running in user-space or executing a system call in the kernel, in which case it may be blocked. Unless we allow intrusive changes to the kernel code, signaling a process is the only method to force a process from user-space into the kernel or to interrupt a blocking system call. The `SIGSTOP` signal is guaranteed to be delivered and not ignored or blocked by the process. Using signals simplifies quiescing as signal delivery already handles potential race conditions, particularly in the presence of threads.

Using `SIGSTOP` to force processes into the stopped state has additional benefits for processes that are running or blocked in the kernel, which will handle the `SIGSTOP` immediately prior to returning to user mode. If a process is in a non-interruptible system call or handling an interrupt or trap, it will be quiesced after the kernel processing of the respective event. The processing time for these events is generally small. If a process is in an interruptible system call, it will immediately return and handle the signal. The effect of the signal is transparent as the system call will in most cases be automatically restarted, or in some cases return a suitable error code that the caller should be prepared to handle. The scheme is elegant in that it builds nicely on the existing semantics of Unix/Linux, and ideal in that it forces processes to a state with only a trivial kernel stack to save and restore on checkpoint-restart.

In quiescing a pod, we must be careful to also handle potential side effects [27] that can occur when a signal is sent to a process. For example, the parent of a process is always notified by a signal when either `SIGSTOP` or `SIGCONT` signals are handled by the process, and a process that is traced always notifies the tracer process about every signal received. While these signals can normally occur on a Unix system, they may have undesirable side effects in the context of checkpoint-restart. We address this issue by ensuring that the virtualization layer masks out signals that are generated as a side effect of the quiesce and restore operations.

The use of `SIGSTOP` to quiesce processes is sufficiently generic to handle every execution scenario with the exception of three cases in which a process may already be in a state similar to the stopped state. First, a process that is already stopped does not need to be quiesced, but instead needs to be marked so that the restart correctly leaves it in the stopped state instead of sending it a `SIGCONT` to resume execution.

Second, a process executing the `sigsuspend` system call is put in a deliberate suspended state until it receives a signal from a given set of signals. If a process is blocked in that system call and then checkpointed, it must be accounted for on restart by having the restarting process call `sigsuspend` as the last step of the restart, instead of stopping itself. Otherwise, it will resume to user mode without really having received a valid signal.

Third, a process that is traced via the `ptrace` mechanism [13] will be stopped for tracing at any location where a trace event may be generated, such as entry and exit of system calls, receipt of signals, events like fork, vfork, exec, and so forth. Each such trace point generates a notification to the controlling process. The ptrace mechanism raises two issues. First, a `SIGSTOP` that is sent to quiesce a pod will itself produce a trace event for traced processes, which—while possible in Unix—is undesirable from a look-and-feel point of view (imagine your debugger reporting spurious signals received by the program). This is solved by making traced process exempt from quiesce (as they already are stopped) and from continue (as they should remain stopped). Second, like `sigsuspend`, the system must record at which point the process was traced, and use this data upon restart. The action to be taken at restart varies with the specific trace event. For instance, for system call entry, the restart code will not stop the pro-

cess but instead cause it to enter a ptrace-like state in which it will block until told to continue. Only then will it invoke the system call directly, thus avoiding an improper trigger of the system call entry event.

6 Process Forest

To checkpoint multiple cooperating processes, it is crucial to capture a globally consistent state across all processes, and preserve process dependencies. Process dependencies include *process hierarchy* such as parent-child relationships, identifiers that define *process relationships* such as group and session identifiers (PGIDs and SIDs respectively), and *shared resources* such as common file descriptors. The first two are particularly important for interactive applications and other activities that involve job control. All of these dependencies must be checkpointed and restarted correctly. The term *process forest* encapsulates these three components: hierarchy, relationships and resources sharing. On restart, the restored process forest must satisfy all of the constraints imposed by process dependencies. Otherwise, applications may not work correctly. For instance, incorrect settings of SIDs will cause incorrect handling of signals related to terminals (including `xterm`), as well as confuse job control since PGIDs will not be restored correctly either.

A useful property of our checkpoint-restart algorithm is that the restart phase can recreate the process forest using standard system calls, simplifying the restart process. However, system calls do not allow process relationships and identifiers to be changed arbitrarily after a process has already been created. A key observation in devising suitable algorithms for saving and restoring the process forest is determining what subset of dependencies require a priori resolution, then leaving others to be setup retroactively.

There are two primary process relationships that must be established as part of process creation to correctly construct a process forest. The key challenge is preserving session relationships. Sessions must be inherited by correctly ordering process creation because the operating system interface only allows a process to change its own session, to change it just once, and to change it to a new session and become the leader. The second issue is preserving thread group relationships, which arises in Linux because of its threading model which treats threads as special processes; this issue does not arise in operating system implementations which do not treat threads as processes. Hereinafter we assume the threading model of Linux 2.6 in which threads are grouped into thread groups with a single thread group leader, which is always the first thread in the group. A thread must be created by its thread group leader because the operating system provides no other way to set the thread group. Given the correct handling of session identifiers and thread groups, other relationships and shared resources can be manipulated after process creation using the operating system interface, and are hence simply assigned once all processes have been created.

Since these two process relationships must be established at process creation time to correctly construct a process forest, the order in which processes are created is crucial. Simply reusing the parent-child relationships maintained by the kernel to create a matching process forest is not sufficient since the forest depends on more than the process hierarchy at the time of checkpoint. For example, it is important to know the original parent of a process to ensure that it inherits its correct SID, however since orphaned children are promptly re-parented to `init`, the information about their original parent is lost. While one could log all process creations and deletions to later determine the original parent, this adds unnecessary runtime overhead and complexity.

We introduce two algorithms—`DumpForest` and `MakeForest`—that use existing operating system interfaces to efficiently save and restore the process forest, respectively. The algorithms correctly restore a process forest at restart that is the same as the original process forest at checkpoint. However, they do not require any state other than what is available at checkpoint time because they do not necessarily recreate the matching process forest in the same way as the original forest was created.

6.1 DumpForest Algorithm

The `DumpForest` algorithm captures the state of the process forest in two passes. It runs in linear time with the number of process identifiers in use in a pod. A process identifier is in use even if a process has terminated as long as the identifier is still being used, for example as an SID for some session group. The first pass scans the list of processes identifiers within the pod and fills in a table of entries; the table is not sorted. Each entry in the table represents a PID in the forest. The second pass records the process relationships by filling in information in each table entry. A primary goal of this pass is to determine the creating parent (*creator*) of each process, including which processes have `init` as their parent. At restart, those processes will be created first to serve as the roots of the forest, and will recursively create the remaining processes as instructed by the table.

Each entry in the table consists of the following set of fields: status, PID, SID, thread group identifier, and three pointers to the table locations of the entry's creator, next sibling, and first child processes to be used by `MakeForest`. Note that these processes may not necessarily correspond to the parent, next sibling, and first child processes of a process at the time of checkpoint. Table 1 lists the possible flags for the status field. In particular, Linux allows a process to be created by its sibling, thereby

inheriting the same parent, which differs from traditional parent-child only fork creation semantics; a `Sibling` flag is necessary to note this case.

Flag	Property of Table Entry
Dead	Corresponds to a terminated process
Session	Inherits ancestor SID before parent changes its own
Thread	A thread but not a thread group leader
Sibling	Created by sibling via parent inheritance

Table 1: Possible flags in the status field

6.1.1 Basic Algorithm

For simplicity, we first assume no parent inheritance in describing the `DumpForest` algorithm. The first pass of the algorithm initializes the PID and SID fields of each entry according to the process it represents, and all remaining fields to be empty. As shown in Algorithm 1, the second pass calls `FindCreator` on each table entry to populate the empty fields and alter the status field as necessary. The algorithm can be thought of as determining under what circumstances the current parent of a process at time of checkpoint cannot be used to create the process at restart.

The algorithm looks at each table entry and determines what to do based on the properties of the entry. If the entry is a thread and not the thread group leader, we mark its creator as the thread group leader and add `Thread` to its status field so that it must be created as a thread on restart. The thread group leader can be handled as a regular process, and hence is treated as part of the other cases.

Otherwise, if the entry is a session leader, this is an entry that at some point called `setsid`. It does not need to inherit its session from anyone, so its creator can just be set to its parent. If a pod had only one session group, the session leader would be at the root of the forest and its parent would be `init`.

Otherwise, if the entry corresponds to a dead process (no current process exists with the given PID), the only constraint that must be satisfied is that it inherit the correct session group from its parent. Its creator is just set to be its session leader. The correct session group must be maintained for a process that has already terminated because it may be necessary to have the process create other processes before terminating itself, to ensure that those other processes have their session groups set correctly.

Otherwise, if the entry corresponds to an orphan process, it cannot inherit the correct session group from `init`. Therefore, we add a placeholder process in the table whose function on restart is to inherit the session group from the entry's session leader, create the process, then terminate so that the process will be orphaned. The placeholder is assigned an arbitrary PID that is not already in the table, and the SID identifying the session. To remember to terminate the placeholder process, the placeholder entry's status field is marked `Dead`.

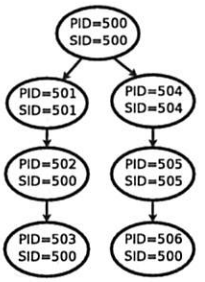
Algorithm 1 DumpForest (second pass)

```
 1: Procedure DumpForest
 2: for all entries ent in the table do
 3:   if (ent.creator == NIL) then
 4:     call FindCreator(ent)
 5:   end if
 6: end for
 7: End
 8:
 9: Procedure FindCreator(ent)
10: leader ← session leader entry
11: if ent is a dead process then
12:   parent ← init
13:   ent.status |= Dead
14: else
15:   parent ← parent process entry
16: end if
17: if ent is thread (but not thread group leader) then
18:   ent.creator ← thread group leader
19:   ent.status |= Thread
20: else if (ent == leader) then
21:   ent.creator ← parent
22: else if (ent.status & Dead) then
23:   ent.creator ← leader
24: else if (parent == init) then
25:   call AddPlaceHolder(ent, leader)
26: else if (ent.sid == parent.sid) then
27:   ent.creator ← parent
28: else
29:   ent.creator ← parent
30:   sid ← ent.sid
31:   repeat
32:     ent.status |= Session
33:     if (ent.creator == init) then
34:       call AddPlaceHolder(ent, leader)
35:     end if
36:     ent ← ent.creator
37:     if (ent.creator == NIL) then
38:       call FindCreator(ent)
39:     end if
40:   until (ent.sid == sid) or (ent.status & Session)
41: end if
42: End
43:
44: Procedure AddPlaceHolder(ent, leader)
45: add new entry new to table
46: new.creator ← leader
47: new.status |= Dead
48: ent.creator ← new
49: End
```

(Note: when the creator field is set, the matching child and sibling fields are adjusted accordingly; details are omitted for brevity).

Otherwise, if the entry's SID is equal to its parent's, the only constraint that must be satisfied is that it inherit the correct session group from its parent. This is simply done by setting its creator to be its parent.

If none of the previous cases apply, then the entry corresponds to a process which is not a session leader, does not have a session group the same as its parent, and therefore whose session group must be inherited from an ancestor further up the process forest. This case arises because the process was forked by its parent before the parent changed

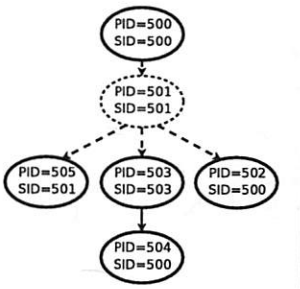


(a) Scenario

PID	Creator	Status
500	init	
501	500	
502	501	*Session*
503	502	
504	500	
505	504	*Session*
506	505	*Session*

(b) Resulting table

Figure 1: Simple process forest



(a) Scenario

PID	Creator	Status
500	init	
501	init	
997	500	*Dead*
502	997	
998	500	*Dead*
503	998	
504	503	*Session*
999	501	*Dead*
505	999	

(b) Resulting table

Figure 2: Process forest with deletions

its own SID. Its creator is set to be its parent, but we also mark its status field `Session`, indicating that at restart the parent will need to fork before (potentially) creating a new session. When an entry is marked with `Session`, it is necessary to propagate this attribute up its ancestry hierarchy until an entry with that session group is located. In the worst case, this would proceed all the way to the session leader. This is required for the SID to correctly descend via inheritance to the current entry. Note that going up the tree does not increase the runtime complexity of the algorithm because traversal does not proceed beyond entries that already possess the `Session` attribute.

If the traversal fails to find an entry with the same SID, it will stop at an entry that corresponds to a leader of another session. This entry must have formerly been a descendant of the original session leader. Its creator will have already been set `init`. Because we now know that it needs to pass the original SID to its own descendants, we re-parent the entry to become a descendant of the original session leader. This is done using a placeholder in a manner similar to how we handle orphans that are not session leaders.

6.1.2 Examples

Figure 1 illustrates the output of the algorithm on a simple process forest. Figure 1a shows the process forest at checkpoint time. Figure 1b shows the table generated by `DumpForest`. The algorithm first creates a table of seven entries corresponding to the seven processes, then proceeds to determine the creator of each entry. Processes 502, 505, and 506 have their `Session` attributes set, since they must be forked off *before* their parents' session identifiers are changed. Note that process 505 received this flag by propagating it up from its child process 506.

Figure 2 illustrates the output of the algorithm on a process forest with a missing process, 501, which exited before the checkpoint. Figure 2a shows the process forest at checkpoint time. Figure 2b shows the table generated by `DumpForest`. While the algorithm starts with six entries in the table, the resulting table has nine entries since three placeholder processes, 997, 998, and 999, were added to maintain proper process relationships. Observe that process 503 initially has its creator set to `init`, but is reparented to the placeholder 998 as part of propagating its child's `Session` attribute up the tree.

6.1.3 Supporting Linux Parent Inheritance

We now discuss modifications to the basic `DumpForest` algorithm for handling the unique parent inheritance feature in Linux which allows a process to create a sibling. To inherit session relationships correctly, parent inheritance must be accounted for in determining the creators of processes that are not session leaders.

If its session leader is alive, we can determine that a process was created by its sibling if its parent is the creator of the session leader. If the session leader is dead, this check will not work since its parent is now `init` and there is no longer any information about its original parent. After a process dies, there is no easy way to determine its original parent in the presence of parent inheritance.

To provide the necessary information, we instead record the session a process inherits when it is created, if and only if the process is created with parent inheritance and it is a sibling of the session group leader. This *saved-SID* is stored as part of the process's virtualization data structure so that it can be used later if the process remains alive when the forest needs to be saved. A process created with parent inheritance is a sibling of the session group leader if either its creator is the session group leader, or its creator has the same saved-SID recorded, since the sibling relationship is transitive.

To support parent inheritance, we modify Algorithm 1 by inserting a new conditional after the check for whether an entry's SID is equal to its parent's and before the final else clause in `FindCreator`. The conditional examines whether an entry's saved-SID has been set. If it has been set and there exists another entry in the process forest table whose PID is equal to this saved-SID, the entry's status field is marked `Sibling` so that it will be created with parent inheritance on restart. The entry's creator is set to the entry that owns that PID, which is leader of the

session identified by the saved-SID. Finally, the creator of this session leader is set to be the parent of the current process, possibly re-parenting the entry if its creator had already been set previously.

6.2 MakeForest Algorithm

Given the process forest data structure, the `MakeForest` algorithm is straightforward, as shown in Algorithm 2. It reconstructs the process hierarchy and relationships by executing completely in user mode using standard system calls, minimizing dependencies on any particular kernel internal implementation details. The algorithm runs in linear time with the number of entries in the forest. It works in a recursive manner by following the instructions set forth by the process forest data structure. `MakeForest` begins with a single process that will be used in place of `init` to fork the processes that have `init` set as their creator. Each process then creates its own children.

The bulk of the algorithm loops through the list of children of the current process three times during which the children are forked or cleaned up. Each child that is forked executes the same algorithm recursively until all processes have been created. In the first pass through the list of children, the current process spawns children that are marked `Session` and thereby need to be forked before the current session group is changed. The process then changes its session group if needed. In the second pass, the process forks the remainder of the children. In both passes, a child that is marked `Thread` is created as a thread and a child that is marked `Sibling` is created with parent inheritance. In the third pass, terminated processes and temporary placeholders are cleaned up. Finally, the process either terminates if it is marked `Dead` or calls `RestoreProcessState()` which does not return. `RestoreProcessState()` restores the state of the process to the way it was at the time of checkpoint.

Algorithm 2 MakeForest

```
 1: Procedure MakeForest
 2: for all entries ent in the table do
 3:   if ent.creator == init then
 4:     call ForkChildren(ent)
 5:   end if
 6: end for
 7: End
 8:
 9: Procedure ForkChildren(ent)
10: for all children cld of ent do
11:   if (cld.status & Session) then
12:     call ForkChild(cld)
13:   end if
14: end for
15: if (ent.sid == ent.pid) then
16:   call setsid()
17: end if
18: for all children cld of ent do
19:   if ¬(cld.status & Session) then
20:     call ForkChild(cld)
21:   end if
22: end for
23: for all children cld of ent do
24:   if (cld.status & Dead) then
25:     call waitpid(cld.pid)
26:   end if
27: end for
28: if (ent.status & Dead) then
29:   call exit()
30: else
31:   call RestoreProcessState()
32: end if
33: End
34:
35: Procedure ForkChild(cld)
36: if (cld.status & Thread) then
37:   pid = fork_thread()
38: else if (cld.status & Sibling) then
39:   pid = fork_sibling()
40: else
41:   pid = fork()
42: end if
43: if pid == 0 then
44:   call ForkChildren(cld)
45: end if
46: End
```

7 Shared Resources

After the process hierarchy and relationships have been saved or restored, we process operating system resources that may be shared among multiple processes. They are either *globally shared* at the pod level, such as IPC identifiers and pseudo terminals, or *locally shared* among a subset of processes, such as virtual memory, file descriptors, signal handlers and so forth. As discussed in Section 4, globally shared resources are processed first, then locally shared resources are processed. Shared resources may be referenced by more than one process, yet their state need only be saved once. We need to be able to uniquely identify each resource, and to do so in a manner independent of the operating system instance to be able to restart on another instance.

Every shared resource is represented by a matching kernel object whose kernel address provides a unique identifier of that instance within the kernel. We represent each resource by a tuple of the form ≪`Address`,`Tag`≫, where `address` is its kernel address, and `tag` is a serial number that reflects the order in which the resources were encountered (counting from 1 and on). Tags are, therefore, unique logical identifiers for resources. The tuples allow the same resource representation to be used for both checkpoint and restart mechanisms, simplifying the overall implementation. During checkpoint and restart, they are stored in an associative memory in the kernel, enabling fast translation between physical and logical identifiers. Tuples are registered into the memory as new resources are discov-

ered, and discarded once the entire checkpoint (or restart) is completed. This memory is used to decide whether a given resource (physical or logical for checkpoint or restart respectively) is a new instance or merely a reference to one already registered. Both globally and locally shared resources are stored using the same associative memory.

During checkpoint, as the processes within the pod are scanned one by one, the resources associated with them are examined by looking up their kernel addresses in the associative memory. If the entry is not found (that is, a new resource has been detected) we allocate a new (unique) tag, register the new tuple and record the state of that resource. The tag is included as part of that state. On the other hand, if the entry is found, it means that the resource is shared and has been already dealt with earlier. Hence it suffices to record its tag for later reference. Note that the order of the scan is insignificant.

During restart, the algorithm restores the state of the processes and the resources they use. The data is read in the same order as has been written originally, ensuring that the first occurrence of each resource is accompanied with its actual recorded state. For each resource identifier, we examine whether the tag is already registered, and if not we create a new instance of the required resource, restore its state from the checkpoint data, and register an appropriate tuple, with the address field set to the kernel address that corresponds to the new instance. If a tuple with the specified tag is found, we locate the corresponding resource with the knowledge of its kernel address as taken from the tuple.

Nested shared objects Nested sharing occurs in the kernel when a common resource is referenced by multiple distinct resources rather than by processes. One example are objects that represent a `FIFO` in the filesystem, as a `FIFO` is represented by a single inode which is in turn pointed to by file descriptors of reader and writer ends. Another example is a single backing file that is mapped multiple times within distinct address spaces. In both examples shared objects—file descriptors and address spaces respectively—refer to a shared object, yet may themselves be shared by multiple processes.

Nested sharing is handled similarly to simple sharing. To ensure consistency we enforce an additional rule, namely that a nested object is always recorded prior to the objects that point to it. For instance, when saving the state of a file descriptor that points to a `FIFO`, we first record the state of the `FIFO`. This ensures that the tuples for the nested resource exist in time for the referring object.

Compound shared objects Many instances of nested objects involve a pair of coupled resources. For example, a single pipe is represented in the kernel by two distinct inodes that are coupled in a special form, and Unix domain sockets can embody up to three disparate inodes for the listening, accepting and connecting sockets. We call such objects *compound objects*. Unlike unrelated resources, compound objects have two or more internal elements that are created and interlinked with the invocation of the appropriate kernel subroutine(s) such that their lifespans are correlated, e.g. the two inodes that constitute a pipe.

We consistently track a compound object by capturing the state of the entire resource including all components at once, at the time it is first detected, regardless of through which component it was referred. On restart, the compound object will be encountered for the first time through some component, and will be reconstructed in its entirety, including all other components. Then only the triggering component (the one that was encountered) will need to be attached to the process that owns it. The remaining components will linger unattached until they are claimed by their respective owners at a later time.

The internal ordering of the elements that compose a compound object may depend on the type of the object. If the object is symmetric, such as `socketpairs`, its contents may be saved at an arbitrary order. Otherwise, the contents are saved in a certain order that is particularly designed to facilitate the reconstruction of the object during restart. For example, the order for pipes is first the read-side followed by the write-side. The order for Unix domain sockets begins with the listening socket (if it exists), followed by the connecting socket and finally the accepting socket. This order reflects the sequence of actions that is required to rebuild such socket-trios: first create a listening socket, then a socket that connects to it, and finally the third socket by accepting the connection.

Memory sharing Since memory footprint is typically the most dominant factor in determining the checkpoint image size, we further discuss how recording shared resources is done in the case of memory. A memory region in a process's address space can be classified along two dimensions, one is whether it is mapped to a backing file or anonymous, and the other is whether it is private to some address space or shared among multiple ones. For example, text segments such as program code and shared libraries are mapped and shared, IPC shared memory is anonymous and shared, the data section is mapped and private, and the heap and stack are anonymous and private.

Memory sharing can occur in any of these four cases. Handling regions that are shared is straightforward. If a region is mapped and shared, it does not need to be saved since its contents are already on the backing file. If a region is anonymous and shared, it is treated as a normal shared object so that its contents are only saved once. Handling regions that are private is more subtle. While it appears contradictory to have memory sharing with private memory regions, sharing occurs due to the kernel's COW optimization. When a process forks, the kernel defers the creation of a separate copy of the pages for the newly created process until one of the processes sharing the common

Name	Description
apache	apache 2.0.55 with 50 threads (default) loaded w/ `httperf 0.8` (rate=1500, num-calls=20)
make	compilation (make -j 5) of Linux kernel tree
mysql	MySQL 4.2.21 loaded w/ standard `sql-bench`
volano	VolanoMark 2.5 w/ Blackdown Java 1.4.2
UML	User Mode Linux w/ 128 MB and Debian 3.0
gnome-base	Gnome 2.8 session with THINC server
gnome-firefox	gnome-base and Firefox 1.04 with 2 browser windows and 3 open tabs in each
gnome-mplayer	gnome-base and MPlayer 1.0pre7-3.3.5 playing an MPEG1 video clip
Microsoft-office	gnome-base and CrossOver Office 5.0 running Microsoft Office XP with 2 Word documents and 1 Powerpoint slide presentation open

Table 2: Application scenarios

memory attempts to modify it. During checkpoint, each page that has been previously modified and belongs to a private region that is marked COW is treated as a nested shared object so that its contents are only saved once. During restart, the COW sharing is restored. Modified pages in either anonymous and private regions or mapped and private regions are treated in this manner.

8 Experimental Results

To demonstrate the effectiveness of our approach, we have implemented a checkpoint-restart prototype as a Linux kernel module and associated user-level tools and evaluated its performance on a wide range of real applications. We also quantitatively compared our prototype with two other commercial virtualization systems, OpenVZ and Xen. OpenVZ provides another operating system virtualization approach for comparison, while Xen provides a hardware virtualization approach for comparison. We used the latest versions of OpenVZ and Xen that were available at the time of our experiments.

The measurements were conducted on an IBM HS20 eServer BladeCenter, each blade with dual 3.06 GHz Intel Xeon^{TM} CPUs, 2.5 GB RAM, a 40 GB local disk, and QLogic Fibre Channel 2312 host bus adapters. The blades were interconnected with a Gigabit Ethernet switch and linked through Fibre Channel to an IBM FastT500 SAN controller with an Exp500 storage unit with ten 70 GB IBM Fibre Channel hard drives. Each blade used the GFS cluster filesystem [25] to access a shared SAN. Unless otherwise indicated, the blades were running Debian 3.1 distribution and the Linux 2.6.11.12 kernel.

Table 2 lists the nine application scenarios used for our experiments. The scenarios were running an Apache web server, a kernel compile, a MySQL database server, a volano chat server, an entire operating system at user-level using UML, and four desktop applications scenarios run using a full Gnome X desktop environment with an XFree86 4.3.0.1 server and THINC [1] to provide remote display access to the desktop. The four desktop scenarios were running a baseline environment without additional applications, a web browser, a video player, and a Microsoft Office suite using CrossOver Office. The UML scenario shows the ability to checkpoint and restart an entire operating system instance. The Microsoft Office scenario shows the ability to checkpoint and restart Windows applications using CrossOver Office on Linux.

We measured checkpoint-restart performance by running each of the application scenarios and taking a series of ten checkpoints during their execution. We measured the checkpoint image sizes, number of processes that were checkpointed, checkpoint times, and restart times, then averaged the measurements across the ten checkpoints for each application scenario. Figures 3 to 8 show results for our checkpoint-restart prototype.

Figure 3 shows the average total checkpoint image size, as well as a breakdown showing the amount of data in the checkpoint image attributable to the process forest. The total amount of state that is saved is modest in each case and varies according to the applications executed, ranging from a few MBs on most applications to tens of MBs for graphical desktop sessions. The results show that the total memory in use within the pod is the most prominent component of the checkpoint image size, accounting for over 99% of the image size.

An interesting case is UML, that uses memory mapping to store guest main memory using an unlinked backing file. This file is separate from memory and amounts to 129 MB. By using the optimization for unlinked files as discussed in Section 4 and storing the unlinked files separately on the filesystem, the UML state stored in the checkpoint image can be reduced to roughly 1 MB. The same occurs for CrossOver Office, which also maps additional 16 MB of memory to an unlinked backing file.

Figure 4 shows the average number of processes running within the pod at checkpoints for each application scenario. On average the process forest tracks 35 processes in most scenarios, except for `apache` and `volano` with 169 and 839 processes each, most of which are threads. As Figure 3 shows the process forest always occupies a small fraction of the checkpoint, even for `volano`.

Figure 5 shows the average *total* checkpoint times for each application scenario, which is measured from when the pod is quiesced until the complete checkpoint image is written out to disk. We also show two other measures. Checkpoint *downtime* is the time from when the pod is quiesced until the pod can be resumed; it is the time to record the checkpoint data without committing it to disk. *Sync* checkpoint time is the total checkpoint time plus the time to force flushing the data to disk. Average total checkpoint times are under 600 ms for all application scenarios and as small as 40 ms, which is the case for UML. Comparing

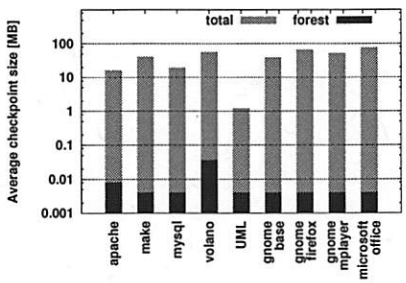


Figure 3: Average checkpoint size

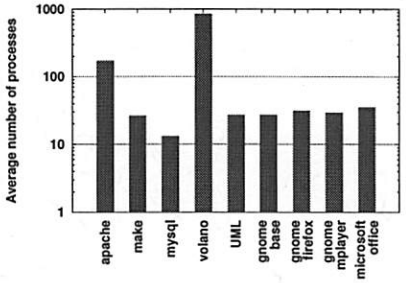


Figure 4: Average no. of processes

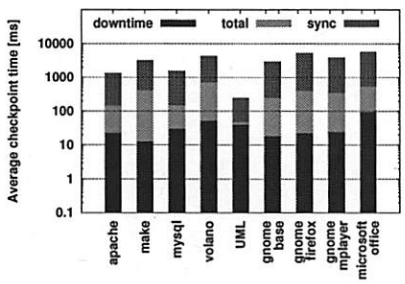


Figure 5: Average checkpoint time

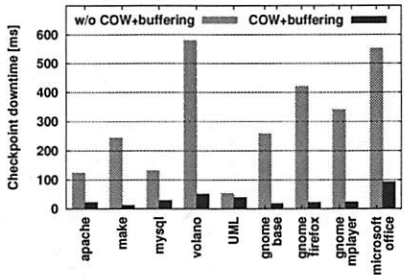


Figure 6: COW and buffering impact

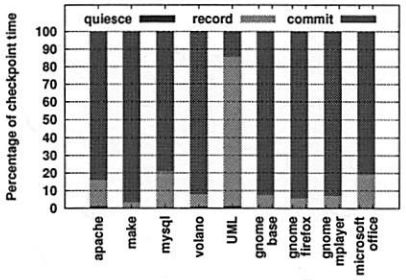


Figure 7: Checkpoint time breakdown

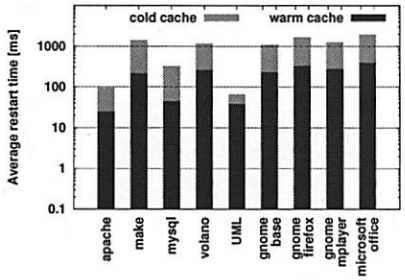


Figure 8: Average restart time

with Figure 3, the results show that both the total checkpoint times and the sync times are strongly correlated with the checkpoint sizes. Writing the filesystem, particularly with forced flushing of the data to disk, is largely limited by the disk I/O rate. For example, `gnome-base` has an average checkpoint size of 39 MB and an average sync checkpoint time of just under 3 s. This correlates directly with the sustained write rate for GFS, which was roughly 15 MB/s in our measurements.

Perhaps more importantly, checkpoint downtimes in Figure 5 show that the average time to actually perform the checkpoint without incurring storage I/O costs is small, ranging from 12 ms for a kernel make to at most 90 ms for a full fledged desktop running Microsoft Office. Though an application is unresponsive while it is quiesced and being checkpointed, even the largest average checkpoint downtimes are less 100 ms. Furthermore, the average checkpoint downtimes were less than 50 ms for all application scenarios except Microsoft Office.

Figure 6 compares the checkpoint downtime for each application scenario with and without the memory buffering and COW mechanisms that we employ. Without these optimizations, checkpoint data must be written out to disk before the pod can be resumed, resulting in checkpoint downtimes that are close to the total checkpoint times shown in Figure 5. The memory buffering and COW checkpoint optimization reduce downtime from hundreds of milliseconds to almost always under 50 ms, in some cases even as much as an order of magnitude.

Figure 7 shows the breakdown of the total checkpoint time (excluding sync) for each application scenario, as percentage of the total time attributable to different steps: *quiesce*—the time to quiesce the pod, *record*—the time to record the checkpoint data, and *commit*—the time to commit the data by writing it out to storage. The commit step amounts to 80-95% of the total time in almost all application scenarios, except for UML where it amounts to only 15% due to a much smaller checkpoint size. Quiescing the processes took less then 700 µs for all application scenarios except `apache` and `volano`, which took roughly 1.5 ms and 5 ms, respectively. The longer quiesce times are due to the large number of processes being executed in `apache` and `volano`. The time to generate and record the process forest was even smaller, less than 10 µs for all applications except `apache` and `volano`, which took 30 µs and 336 µs respectively. The time to record globally shared resources was under 10 µs in all cases.

Figure 8 presents the average total restart times for each application scenario. The restart times were measured for two distinct configurations: *warm cache*—restart was done with a warm filesystem cache immediately after the checkpoint was taken, *cold-cache*—restart was done with a cold filesystem cache after the system was rebooted, forcing the system to read the image from the disk. Warm cache restart times were less than .5 s in all cases, ranging from 24 ms for `apache` to 386 ms for a complete Gnome desktop running Microsoft Office. Cold cache restart times were longer as restart becomes limited by the disk I/O rate. Cold cache restart times were less than 2 s in all cases, ranging from 65 ms for UML to 1.9 s for Microsoft Office. The cold restart from a checkpoint image is still noticeably faster than the checkpoint to the filesystem with flushing

because GFS filesystem read performance is much faster than its write performance.

To provide a comparison with another operating system virtualization approach, we also performed our experiments with OpenVZ. We used version 2.6.18.028stab on the same Linux installation. Because of its lack of GFS support, we copied the installation to the local disk to conduct experiments. Since this configuration is different from what we used with our prototype, the measurements are not directly comparable. However, they provide some useful comparisons between the two approaches. We report OpenVZ results for `apache`, `make`, `mysql` and `volano`; OpenVZ was unable to checkpoint the other scenarios. Table 3 presents the average total checkpoint times, warm cache restart times, and checkpoint image sizes for these applications. We ignore sync checkpoint times and cold cache restart times to reduce the impact of the different disk configurations used.

Scenario	Checkpoint [s]	Restart [s]	Size [MB]
apache	0.730	1.321	7.7
make	2.230	1.376	53
mysql	1.793	1.288	22
volano	2.036	1.300	25

Table 3: Checkpoint-restart performance for subset of applications that worked on OpenVZ

The results show that OpenVZ checkpoint and restart times are significantly worse than our system. OpenVZ checkpoint times were 5.2, 5.6, 12.4, and 3.0 times slower for `apache`, `make`, `mysql` and `volano`, respectively. OpenVZ restart times were 55.0, 6.6, 29.9, and 5.0 times slower for `apache`, `make`, `mysql` and `volano`, respectively. OpenVZ checkpoint sizes were .48, 1.3, 1.2, and .46 times the sizes of our system. The difference in checkpoint sizes was relatively small and does not account for the huge difference in checkpoint-restart times even though different filesystem configurations were used due to OpenVZ's lack of support for GFS. OpenVZ restart times did not vary much among application scenarios, suggesting that container setup time may constitute a major component of latency.

To provide a comparison with a hardware virtualization approach, we performed our experiments with Xen. We used Xen 3.0.3 with its default Linux 2.6.16.29. We were unable to find a GFS version that matched this configuration, so we used the local disk to conduct experiments. We also used Xen 2.0 with Linux 2.6.11 because this configuration worked with GFS. In both cases, we used the same kernel for both "dom0" and "domU". We used three VM configurations with 128 MB, 256 MB, and 512 MB of memory. We report results for `apache`, `make`, `mysql`, UML, and `volano`; Xen was unable to run the other scenarios due to lack of support for virtual consoles. Table 4 presents the average total checkpoint times, warm cache restart times, and checkpoint image sizes for these applications. We report a single number for each configuration instead of per application since Xen results were directly correlated with the VM memory configuration and did not depend on the applications scenario. Checkpoint image size was determined by the amount of RAM configured. Checkpoint and restart times were directly correlated with the size of the checkpoint images.

Xen Config.	Checkpoint [s]		Restart [s]		Image Size [MB]
	Xen 3	Xen 2	Xen 3	Xen 2	
128 MB	3.5	5.5	1.6	0.8	129
256 MB	10.3	12	13.4	6.6	257
512 MB	25.9	19	27.3	12	513

Table 4: Checkpoint-restart performance for Xen VMs

The results show that Xen checkpoint and restart times are significantly worse than our system. Xen 3 checkpoint times were 5.2 (`volano` on 128 MB) to 563 (UML on 512 MB) times slower. Xen 3 restart times were 6.2 (`volano` on 128 MB) to 1137 (`apache` on 512 MB) slower. Xen results are also worse than OpenVZ; both operating system virtualization approaches performed better. Restart times for the 256 MB and 512 MB VM configurations were much worse than the 128 MB VM because the images ended up being too large to be effectively cached in the kernel, severely degrading warm cache restart performance. Note that although precopying can reduce application downtime for Xen migration [4], it will not reduce total checkpoint-restart times.

9 Conclusions

We have designed, implemented, and evaluated a transparent checkpoint-restart mechanism for commodity operating systems that checkpoints and restarts multiple processes in a consistent manner. Our system combines a kernel-level checkpoint mechanism with a hybrid user-level and kernel-level restart mechanism to leverage existing operating system interfaces and functionality as much as possible for transparent checkpoint-restart. We have introduced novel algorithms for saving and restoring extended process relationships and for efficient handling of shared state across cooperating processes. We have implemented a checkpoint-restart prototype and evaluated its performance on real-world applications. Our system generates modest checkpoint image sizes and provides fast checkpoint and restart times without modifying, recompiling, or relinking applications, libraries, or the operating system kernel. Comparisons with two commercial systems, OpenVZ and Xen, demonstrate that our prototype provides much faster checkpoint-restart performance and more robust checkpoint-restart functionality than these other approaches.

Acknowledgments

Dan Phung helped with implementation and experimental results. Eddie Kohler, Ricardo Baratto, Shaya Potter and Alex Sherman provided helpful comments on earlier drafts of this paper. This work was supported in part by a DOE Early Career Award, NSF ITR grant CNS-0426623, and an IBM SUR Award.

References

[1] R. Baratto, L. Kim, and J. Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 277–290, Brighton, UK, Oct. 2005.

[2] A. Beguelin, E. Seligman, and P. Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, June 1997.

[3] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI 2005)*, pages 273–286, Boston, MA, May 2005.

[5] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, Bolton Landing, NY, Oct. 2003.

[6] P. Gupta, H. Krishnan, C. P. Wright, J. Dave, and E. Zadok. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-1, Dept. of Computer Science, Stony Brook University, Jan. 2004.

[7] Y. Huang, C. Kintala, and Y. M. Wang. Software Tools and Libraries for Fault Tolerance. *IEEE Bulletin of the Tecnnical Committee on Operating System and Application Environments*, 7(4):5–9, Winter 1995.

[8] L. V. Kale and S. Krishnan. CHARM++: a Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, pages 91–108, Washington, DC, Sept. 1993.

[9] B. A. Kingsbury and J. T. Kline. Job and Process Recovery in a UNIX-based Operating System. In *Proceedings of the USENIX Winter 1989 Technical Conference*, pages 355–364, San Diego, CA, Jan. 1989.

[10] C. R. Landau. The Checkpoint Mechanism in KeyKOS. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 86–91, Dourdan, France, Sept. 1992.

[11] Linux Software Suspend. `http://www.suspend2.net`.

[12] Linux VServer. `http://www.linux-vserver.org`.

[13] M. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[14] Network Appliance, Inc. `http://www.netapp.com`.

[15] OpenVZ. `http://www.openvz.org`.

[16] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 361–376, Boston, MA, Dec. 2002.

[17] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Dept. of Computer Science, University of Tennessee, July 1997.

[18] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the USENIX Winter 1995 Technical Conference*, pages 213–223, New Orleans, LA, Jan. 1995.

[19] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[20] D. Price and A. Tucker. Solaris Zones: Operating Systems Support for Consolidating Commercial Workloads. In *Proceedings of the 18th Large Installation System Administration Conference*, pages 241–254, Atlanta, GA, Nov. 2004.

[21] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies—a Safe Method to Survive Software Failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 235–248, Brighton, UK, Oct. 2005.

[22] E. Roman. A Survey of Checkpoint/Restart Implementations. Technical Report LBNL-54942, Lawrence Berkeley National Laboratory, July 2002.

[23] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang. Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18*, page 300.2, Washington, DC, Apr. 2005.

[24] B. K. Schmidt. *Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches*. PhD thesis, Stanford University, Aug. 2000.

[25] S. R. Soltis, T. M. Ruwart, and M. T. O'Keefe. The Global File System. In *Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems*, pages 319–342, College Park, MD, Sept. 1996.

[26] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the USENIX 2004 Annual Technical Conference, General Track*, pages 29–44, Boston, MA, June 2004.

[27] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Professional Computing Series. Addison-Wesley, Reading, MA, USA, 1993.

[28] T. Tannenbaum and M. Litzkow. The Condor Distributed Processing System. *Dr. Dobb's Journal*, 20(227):40–48, Feb. 1995.

[29] VMware, Inc. `http://www.vmware.com`.

Reboots are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly

Andrew Baumann* Jonathan Appavoo† Robert W. Wisniewski†
Dilma Da Silva† Orran Krieger† Gernot Heiser*

* *University of New South Wales and National ICT Australia*
† *IBM T.J. Watson Research Center*

Abstract

Patches to modern operating systems, including bug fixes and security updates, and the reboots and downtime they require, cause tremendous problems for system users and administrators. Dynamic update allows an operating system to be patched without the need for a reboot or other service interruption. We have taken the approach of building dynamic update functionality directly into an existing operating system, K42.

To determine the applicability of our update system, and to investigate the changes that are made to OS code, we analysed K42's revision history. The analysis showed that our original system could only support half of the desired changes to K42. The main problem preventing more changes from being converted to dynamic updates was our system's inability to update interfaces. Other studies, as well as our own investigations, have shown that change to interfaces is also prevalent in systems such as Linux. Thus, it is apparent that a dynamic update mechanism needs to handle interface changes to be widely applicable.

In this paper, we describe how to support interface changes in a modular dynamic update system. With this improvement, approximately 79% of past performance and bug fix changes to K42 could be converted to dynamic updates, and we expect the proportion would be even higher if the fixes were being developed for dynamic update. Measurements of our system show that the runtime overhead is very low, and the time to apply updates is acceptable.

This paper makes the following contributions. We present a mechanism to handle interface changes for dynamic updates to an operating system. For performance-sensitive updates, we show how to apply changes lazily. We discuss lessons learned, including how an operating system can be structured to better support dynamic update. We also describe how our approach extends to other systems such as Linux, that although structured modularly, are not strictly object-oriented like K42.

1 Introduction

Patches and updates to modern operating systems are a significant problem for users and administrators. Operating system vendors are releasing an increasing volume of patches at a higher frequency [11], and these patches require restarting services or rebooting the whole system, resulting in downtime that is becoming increasingly costly. This downtime, even if scheduled, is expensive, causing administrators to trade-off its cost against the risks of remaining unpatched. Many do not apply well-announced and widely-propagated security-critical patches for weeks [23]. Furthermore, rebooting a system causes loss of transient state, and thus may be a serious inconvenience to its users.

When we discuss updates, we are considering primarily the kinds of changes that are released in the maintenance process of a mainstream operating system after a major release. For example, bug fixes, security fixes, and performance improvements. Significant new features are rarely released in this way, because application software would need to be updated to take advantage of them. These updates are usually released regularly, depending on their urgency, and are developed and tested by the operating system vendor before distribution to system administrators.

1.1 Existing approaches

Here we give an overview of current approaches to updating systems without loss of service; more closely related work is discussed later, in Section 8.

Traditionally, the solution to the problems of updates and reliability has been to use redundant hardware; that is, either specialised hardware that processes requests on identical machines, or more commonly commodity hardware. If commodity hardware is used, and the service maintains state (unlike a traditional web server), software support must be provided to maintain synchroni-

sation between the redundant systems.

A similar approach is to use virtualisation instead of physically separate hardware. This also requires software support to maintain synchronisation, or a mechanism to migrate applications between virtual machines at update time [22].

Outside the operating system, the most common approach to dynamic updating is to build-in support for updates in a language-specific [3, 10, 14, 20, 25] or domain-specific [1,6] manner. Our work can be seen as a domain-specific approach to achieving dynamic update for operating systems.

1.2 Our approach

Our goal is to provide dynamic update support within the operating system itself, without the need to re-implement or significantly restructure its code. Ideally, it should be possible to load an update into the system similarly to the way we can load kernel modules to add functionality.

We observe that pressures of software development, safety, extensibility, and configurability are driving modern operating systems, even those with a monolithic kernel structure, to become highly modular or componentised. For example, several projects have added more modularity to Linux to enable fault isolation [26, 28]. Software construction techniques such as abstract types, data hiding and encapsulation, and separation of concerns are features of these modular interfaces.

Our approach is to leverage module boundaries to update the code and data within a module without affecting the rest of the system. We provide mechanisms for safely updating a specific module, and transforming the data structures maintained by that module. We repeatedly use those mechanisms to update all modules involved in a larger change, achieving a whole-system update as a series of small self-contained changes. Despite the module-based approach, we are able to apply updates even when a module's interface changes.

1.3 Overview

In previous work [4], we developed a prototype implementation of dynamic update for the K42 research operating system, and tested it with a small number of hand-picked update examples. This prototype is outlined in the following Section 2.

Selecting a small number of changes to convert to dynamic updates, as we had done previously, showed that our system worked but did not help to determine whether the system would be able to apply all of the relevant changes as dynamic updates. To properly assess the coverage of our dynamic update system, we conducted a study of the K42 revision history, that is described in Section 3. In our previous prototype, changes to module interfaces could not be applied as dynamic updates. We had expected such changes to be rare, however the results of our study showed that they were relatively frequent—the majority of new features and a significant proportion of bug fixes and performance improvements included changes to interfaces. Our own investigation and other studies have found that the same is also true for Linux.

To provide a more complete update system, we designed a dynamic update mechanism for an OS that handles interfaces changes, as described in Section 4. This section also describes the new lazy update functionality we have developed for mitigating the performance impact of large updates. Revisiting the revision history analysis with these improvements, we found that the majority of all changes, and an even higher proportion of maintenance changes could now be converted to dynamic updates.

We have measured the performance impact of adding the capability to perform dynamic updates to the system, and found it to be negligible. We have also measured the costs incurred when a dynamic update is applied. These results are presented in Section 5, along with a description of example updates enabled by the added functionality. We then discuss lessons learned from implementing dynamic update in K42 in Section 6, including how we would structure the system differently to better support it.

The work presented here enables dynamic updates to the operating system, but also raises some questions for future research. In Section 7 we discuss open issues in the area, and in Section 8 we describe related and complementary work. Section 9 concludes.

To summarise, our primary contributions over the previous work are as follows. We have conducted a broad analysis of changes in the revision history of an OS, and used this to assess the applicability of our update system. Based on the limitations identified, we have extended the model to support interface changes, which significantly increased the scope of changes that we could support as dynamic updates. We have also added support for lazy conversion of data structures, because the performance impact of converting all data structures at once could be dramatic. Finally, we include an evaluation of the performance characteristics of our system, and a discussion of our experiences using and developing it.

2 Background

We previously developed a prototype dynamic update system [4, 5]. Because that work is essential background information, we briefly summarise it here.

2.1 Design

We identified several fundamental requirements for an operating system to provide dynamic update capability. The most important of these are a modular system structure, a mechanism for detecting a safe point to update a given module, and state tracking and transfer mechanisms to locate and transform the state information maintained by a module.

Given these requirements, the generic update process for a single module is as follows: First, the code associated with an update is loaded into the system by a kernel module loader, or similar mechanism. Next, a *state tracking mechanism* is used to locate all data instances affected by an update. Then, using a level of indirection on module invocations, we block any new accesses to the affected module. Once the *safe point mechanism* detects that the module is idle, or *quiescent*, we update the code in the module and transform its data structures using the *state transfer mechanism*. Finally, having finished the update, the new module is made accessible, and any blocked calls are resumed.

2.2 Dynamic update in K42

K42 is an operating system project targeting scalability and customisability [13]. It runs primarily on 64-bit PowerPC systems, and supports the Linux API and ABI. K42 is implemented in C++, and is object-oriented: each resource managed by the kernel is provided by one or more distinct object instances. To improve scalability in an SMP system, all objects are accessed indirectly through a global *object translation table* (OTT); this indirection also enables dynamic update.

In K42, *state-transfer functions* are implemented for each object, and convert an object's internal state to, or from, a common intermediate representation. Our system detects *quiescence* by tracking the lifetime of kernel threads, and providing a mechanism to determine when all threads that were active when access to an object was blocked have terminated. This works well, because the kernel is event-driven, and its threads are short-lived.

Our original implementation added two features to K42 to support dynamic update: a kernel module loader and a factory mechanism. The module loader is similar to the one used in Linux, but simpler because updated code is only accessed indirectly through object references. The factory mechanism is responsible for the creation, destruction, and tracking of object instances within K42 via the factory design pattern [9]. Factories in K42 are live objects accessed through well-known references, one per class. They allow us to update all the objects affected by a code change, ensure that future instantiations use the updated code, and track when all the objects of a given class have been updated.

Using these foundations, we were able to apply dynamic updates to K42. We hand-picked some interesting changes from the K42 revision history, converted those to dynamic updates, and applied them to the running system.

3 Analysis of CVS history

In this section, we describe a study we have performed of changes from the K42 CVS revision history. Questions we sought to answer included:

- What change types are seen in K42's development?
- What proportion of these changes are bugfixes, security fixes, or performance improvements that would be shipped in maintenance releases?
- How many, and what kind of changes could we apply using our dynamic update mechanism?

The broader question of how operating system code evolves is also not well understood, and is a rich area for further investigation.

3.1 Method

K42 was developed over a period of nine years (the first revision is from March 1997), by around five to ten developers. Hence, there is a lot of revision data in the repository to be examined: 4,814 files and 56,199 revisions in the core modules we examined.[1]

One of the drawbacks of CVS is that it operates only at a file and revision level, and does not track any dependencies between files or directories. Thus, we first had to develop mechanisms and heuristics for extracting independent transactions or changes from CVS revisions. This required two assumptions.

First, we assumed that each commit operation by a developer was a single logical change or feature. This is usually true, but not always. A few developers tended to commit unrelated changes together. This means that we see fewer and larger changes than we should, so our results are pessimistic.

Second, we assumed that after each commit the repository was in a consistent state. That is, it could be expected to compile and run correctly. Obviously developers make mistakes, so this is not always true. However, K42 includes an extensive set of regression tests that developers usually run before committing, so in the majority of cases the assumption was valid.

Given these assumptions, we developed a modified version of the *slurp* tool [16] to process the CVS repository data and import it into a database for further analysis, and used an algorithm described by Zimmermann

and Weißgerber [30] to group related CVS revisions into logical transactions. For each source file revision, we also filtered out all the comments, reformatted the code in a consistent style using an *indent* tool, and computed the differences between the cleaned and reformatted source. This significantly reduced the number of irrelevant changes that needed to be examined.

Using this data, we performed two types of analysis. First, we used automatically-computed contextual information to determine which transactions changed only code inside dynamically-updatable objects and could therefore be developed into dynamic updates. Second, we randomly selected transactions from our sample and manually inspected them to gather more accurate and more detailed information about what types of changes are possible, and specifically what prevents changes from being converted to dynamic updates. Based on that result, we estimated from the overall set of changes what proportion could be converted to dynamic updates.

For both analyses we considered only transactions that altered some kernel code. Specifically, the source differences computed in the final step were non-empty, and at least one of the files modified by the transaction was within the `os/kernel` directory. Apart from some common library code, this directory contains the K42 kernel, including process and memory management, IPC mechanisms, exception handlers, boot code, and Linux glue code. File-systems and device drivers are reused from Linux, and their source is maintained elsewhere.

3.2 Automatic analysis

Most of K42 consists of objects accessed indirectly through the object translation table. However, some parts, such as the exception handlers and parts of the scheduling code, are not accessed indirectly, and therefore are not dynamically updatable. To calculate what proportion of changes could be converted to dynamic updates, it is necessary to determine which changes affect only code inside these dynamically-updatable objects.

We would have liked to determine what functions, data structures, and objects were changed by each transaction in the repository. This implies parsing the code. However, a normal C++ parser would read all the header files included by a particular file; effectively it would require reconstructing the K42 source tree for every transaction, which would be very slow. To avoid this, we wrote a pseudo-parser handling just enough of the language (for example, the *class* keyword, function definitions, and braces) to identify a *program context* for every line of C++ source. A program context is a function or class name, or a special *global* context (used, for example, for preprocessor directives).

Given this contextual information, we identified for each transaction any classes or functions added or deleted by that transaction, and also any that were modified. We then examined every transaction that included a change to kernel code (a total of 3618 transactions), and categorised them based on those that added contexts, those that deleted them, and those that just modified code within existing contexts.

Using a list of classes known to be dynamically updatable, we then identified the transactions that changed only dynamically-updatable code. Our list included some classes that do not yet have state-transfer functions or factories, so are currently not updatable. We included these, because the addition of state-transfer functions and factories is relatively simple (the changes are confined to the class itself), and because we believe that showing the limitations of the model is more meaningful than showing those of the K42 implementation.

We found that 22% of transactions only modified or added methods in dynamically-updatable classes. A number of common problems prevented more transactions from being classified as dynamically-updatable:

- changes to code for testing, tracing, or debugging, that would not be released as updates, and so are irrelevant to our target problem;
- changes to initialisation code that would instead be implemented as part of the state transformation and dynamic update load process;
- changes to simple classes that aren't themselves dynamically updatable, but are encapsulated within dynamically-updatable objects, and so could be updated as part of the surrounding object;
- changes to the global context, such as preprocessor definitions or global declarations, that would be handled differently for a dynamic update.

If we include these, the result rises to 48%. If we exclude changes before 2002, when K42 was in a more developmental phase, the total proportion of dynamically updatable transactions rises to 55%.

Due to the automatic nature of this analysis, these results include a certain amount of noise. For example, some changes were committed to the tree, then reverted because they caused regressions, and later committed again with fixes; these should be counted as only one transaction. Other transactions included, upon closer inspection, multiple independent changes. Many changes performed cleanup actions, such as moving code between header files, splitting or merging classes, and so on.

These examples show the limitations of automatically analysing the revision history. The result is skewed by a large number of changes that would never need to be

developed into dynamic updates, however it gives us a reasonable lower bound for the proportion of dynamically updatable changes. For a more accurate result, we conducted the manual analysis described in the following section.

3.3 Manual sampling

The automatic analysis gave us useful information about the overall proportion of dynamically updatable changes, but these results include all the changes in K42's revision history, and the revision history of an experimental operating system does not mirror what would happen in the maintenance of a released operating system. Changes that happened frequently in K42, such as new features, code cleanups or debugging changes, would not be shipped in maintenance updates. For a more accurate analysis of the applicability of our dynamic update system to the specific types of change in which we are interested, we conducted a manual investigation using a sample of the CVS transactions.

We developed a simple web application allowing a human analyst to examine randomly-selected transactions. For each transaction, the analyst was shown the commit log message and other meta data, the source code differences for affected files, and the list of changed program contexts computed by the previous automatic analysis. The analyst then assigned each transaction to a number of categories, using their knowledge of the K42 code, as well as an understanding of what could be changed by a dynamic update. Our goal was to examine a sufficiently large number of transactions to obtain statistically significant conclusions about the proportion of dynamically updatable changes. In total, we have manually analysed 250 transactions.

Some transactions were considered irrelevant to the analysis, and ignored. These included a change that was reverted and then recommitted later, a small number of transactions that included many unrelated changes, and many changes that were functionally-equivalent such as a reorganisation of header files, changes that only added debugging output, changes to preprocessor directives, and so on. In total, 39% of the transactions that we examined were ignored.

We then looked at the change as a whole, and placed it into one of five categories based on its main purpose: bug fixes, security fixes, minor/maintenance performance improvements, new features, and changes for non-functionally-equivalent cleanup or restructure. Of the non-ignored transactions, 48% were restructuring, 36% added new features, 11% were bug fixes, and the remaining 5% performance improvements. We found no security fixes. Because K42 has been used to date only for research purposes, security holes that would necessitate fixes have not been uncovered. However, because security fixes are a subclass of bug fixes, and tend to be of a small, isolated, and feature-less nature [2], we expect that results for the bug fix category will be a good indicator of our system's support for security updates.

We also examined the code differences and determined what was affected by the change: data structures, interfaces, multiple objects, and library functions (recall that we selected any changes affecting the kernel source code, which could also include changes to user libraries).

Finally, we decided whether the change was convertible to a dynamic update. Of the transactions categorised as bug fixes or performance improvements, which we will refer to as maintenance changes, only 50% could be converted to dynamic updates. Of the non-updatable maintenance changes, 58% were ruled out because they changed interfaces, and the remainder changed non-updatable exception handler code. In the other categories, only 11% of new features and 6% of code restructures could be converted to dynamic updates. These results are shown as the *simple update* case in Figure 2.

3.4 Conclusions

Extending our results from a case-study analysis of the K42 revision history to more general conclusions about the applicability of our dynamic update model is potentially error-prone. The revisions in the main branch of a research operating system do not necessarily reflect the maintenance and update release process of a production system. Nevertheless, our study gives an indication of the updates we can expect to see in systems code. In a production operating system in maintenance mode, we would expect far fewer broad restructures and added features, and a greater proportion of performance and security updates or bug fixes.

We were surprised by the high incidence of changes to interfaces, even among the maintenance updates. An interface change in K42 is any change to the virtual methods defined for a class, that would cause code compiled against the previous definition to behave incorrectly. This includes the addition or deletion of methods, arguments, or changes to types, none of which were supported for dynamic update. We were aware that this was a limitation, but believed that it was not significant, as most updates would not change interfaces. However, our results showed that a surprisingly high proportion of kernel updates did require changes to interfaces.

To verify that this problem was not unique to K42, we inspected recent stable releases of the Linux kernel: versions from 2.6.18.1 to 2.6.18.6 inclusive and version 2.6.19.1. These releases include relatively few changes, the largest uncompressed patch being 250 kilobytes in size, and contain only bug and security fixes. However,

four of the seven versions examined included changes to the prototypes of non-inline kernel functions, confirming the prevalence of interface change in Linux.

Other researchers have reached similar conclusions regarding the need to support interface changes. Neamtiu *et al.* [17] conducted a study of source code evolution in several common open-source programs, including the Linux kernel, with the goal of informing the design of dynamic update systems. They concluded that changes to type definitions and function prototypes were both common enough to be an important feature for a dynamic update system to support. Furthermore, another recent study of collateral evolution in Linux device drivers [21] highlighted the problems associated with changes to interfaces in that system.

Therefore, it is clear that for our dynamic update system to be usable, it must support evolution of interfaces within the kernel. In the following section we will discuss how to address this challenge.

4 Extending to complex dynamic updates

Based on our experiences and results from the previous section, to enable more updates to be applied, and to increase the applicability of our system, we have made a series of improvements to its design and implementation. Here we discuss the most significant: support for interface changes and lazy update. We also describe the process of developing and applying an update.

4.1 Interface changes

Our previous design and implementation did not support changes to object interfaces. As shown in the CVS analysis, this was a serious limitation on the applicability of our system. When an object's interface changes, any calling objects that depend on the interface must also be changed. The obvious solution is to update all affected objects in a single atomic operation, however blocking and updating multiple objects may be unworkable. For complex changes, it effectively requires quiescence across the entire kernel, leading to large delays, and potential deadlock and correctness issues (for example, missed interrupts could cause the system to lockup or crash).

From a closer examination of the changes to interfaces observed in the K42 revision history, we found that most of the changes were relatively minor. These included: adding or renaming functions; removing parameters from functions; and extending the parameter list of existing functions, but providing a default parameter to avoid updating all the existing call points.

Informed by these observations, we decided to use the object adaptor design pattern [9]. Adaptors wrap a class

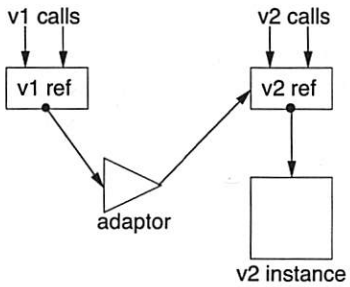


Figure 1: Adaptor object

to make it provide a different interface. In K42, adaptors were implemented for dynamic update, and operate transparently to other objects through the use of the object translation table. They can maintain their own state information, and are able to intercept and rewrite all function calls from old un-updated callers of the object. Changes made possible by adaptors include:

- adjusting virtual method numbers, when functions have been added;
- shuffling parameter registers, and computing or supplying defaults for new parameters;
- altering return values, or directly returning a value (such as an error code) without calling the object.

Not all interface changes can be expressed by an adaptor. In particular, changes that are not backwards-compatible, or where the old interface cannot be provided by operations in the adaptor or on the updated object, are not possible. This includes changes where functionality is removed, or complex restructuring changes, such as when an object's functionality is split into several other objects. However, the forms of interface change supported by adaptors are sufficient for maintenance changes, as will be shown in Section 4.4.

Our design is shown in Figure 1. Any change that alters an object's interface requires an adaptor to be supplied along with the updated code. When the dynamic update is applied, the new object with the updated interface is installed on a new object reference, and an adaptor object is instantiated for the old reference, forwarding calls to the underlying object. Then, caller objects are progressively updated to directly invoke the new interface. Once all old caller objects are updated, as determined by the relevant factory objects, the old reference and the adaptor object are destroyed.

Some low-level code in K42, such as the initial page-fault handlers, is not part of the object system and therefore not updatable. If any of the objects called by such low-level code are updated with an adaptor, then the adaptor will be required permanently, because it is not otherwise possible to update the calling code to use the

new interface. Fortunately, in K42 there is very little code in this category.

The use of adaptor objects follows our fundamental design principle of applying dynamic updates as a series of small independent changes. Adaptors allow us to update the system progressively, without the need to concurrently block access to multiple objects. Adaptors impose additional overhead on all function calls to an affected object, but this overhead is only transient—as the calling objects are updated to versions that support the new interface, which happens as part of the overall update, the adaptors are removed.

4.2 Lazy update

Our original model transformed every object at the time an update was loaded; other dynamic update systems that support changes to data structures have also taken this approach [8, 19]. However, this presents a scalability and performance problem, because some objects may have thousands or more instances present within the kernel; for example, the objects associated with open files or memory regions. When testing a K42 update that altered the in-memory data structures of each open file on a loaded system, we found that the system performance was severely degraded while converting all the affected objects.

To illustrate the scale of this problem, we used the `/proc/slabinfo` file to count instances of different kernel data structures on a moderately-loaded file and compute server running Linux 2.6.18. We found 1.9 million each of the filesystem's *inode* and *vnode* structures, 234,264 blocks in the buffer cache, 51,301 virtual memory areas, and 14,437 open files, to take a few examples. If any of these data structures were changed, it would not be feasible to delay the system's execution while they were all updated.

To address this problem, we implemented the ability to perform the dynamic update lazily [6]. When a lazy update is loaded, affected object references are changed to point to a special lazy-update object. The first time this object is invoked, it initiates the actual update, restarts the method call that triggered it, and then removes itself from the affected object reference. Laziness mitigates the performance impact of updates involving many objects by spreading out the load, because rather than transforming all object instances at once, objects are gradually converted as they are accessed. It achieves this while still guaranteeing that the old code will not be invoked once the initial process of installing the lazy-update objects is complete.

Lazy update also allows us to avoid unnecessarily converting objects that are not invoked between updates or ever again. If an object has been only lazily updated, and another update to that object is loaded, we could use the state-transfer functions from both update versions in sequence, avoiding the cost of twice achieving quiescence in that object. Another modification of the technique would be to combine lazy update with a daemon thread that runs at low priority, updating objects as the system's idle time allows.

4.3 Update process

The changed process for developing and applying a dynamic update, as opposed to the simpler version outlined in Section 2, is as follows. To build a dynamic update, we take the new version of any changed classes, develop and add necessary state-transfer functions, and compile them together with code that initiates the update to form a loadable module. If the update changes a class interface, then an adaptor object must also be implemented and included in the module.

The update module is then loaded into the kernel. Its initialisation code triggers an update of the factories for the affected classes. Next, a new factory walks through the old object instances, either initiating a direct update, or marking them to be updated lazily. Presently, the developer of an update determines whether to use laziness, but this could also be implemented by heuristics in the factories; for example, if there are more than 100 live instances, use lazy update.

To update an individual object, if an adaptor is being used, it is first installed on the old reference, then the state is transferred to the new object while both are quiescent. If an adaptor is not required, the object's reference does not need to change, because all calls conform to the same interface and thus can be intermingled—in this case, the new object simply takes over the old reference.

When an object is updated to understand altered interfaces, its state-transfer function must locate the new reference for any objects whose interfaces have changed. This is done using a special function implemented in the base classes for all objects, that returns the canonical reference for a given object.

As object updates complete, either directly or lazily, the old objects and lazy-update objects are destroyed. Finally, when all old objects of a given type are updated, as determined by the relevant factory, two cleanup operations occur. First, any adaptor objects can be removed and their references reclaimed. Second, the code used for the loaded module corresponding to the previous version of the class, and other static kernel memory associated with that class is reclaimed; however, our module loader implementation does not yet free memory.

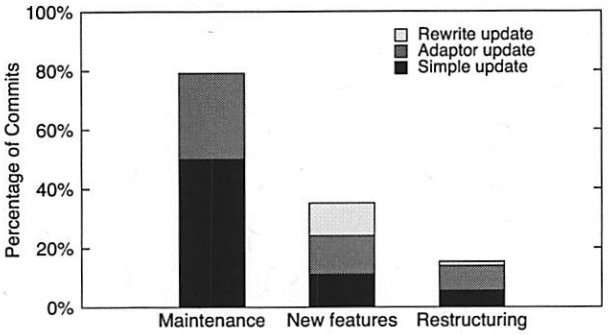


Figure 2: Results of manual CVS analysis

4.4 CVS analysis revisited

With support for limited interface changes in the form of adaptors, we revisited the manual CVS analysis of Section 3. We also considered whether the changes could have been altered slightly to allow them to be dynamically updated. For each transaction, there were now four possibilities: the change could be updated without adaptors, the change could be updated only with an adaptor, the change could be updated only after some simple rewriting of the code (possibly also with an adaptor), or the change could not easily be updated. These results are shown in Figure 2.

We found that all of the maintenance changes that altered interfaces could have been supported through the use of an adaptor, raising the total of updatable maintenance changes to 79% (none required rewriting). Of the other categories, 35% of new features and 15% of code restructures could now be converted to dynamic updates. All of the complex interface changes that could still not be supported by the use of an adaptor were found in the new feature or code restructure categories, confirming that the limited form of interface change supported by adaptors is sufficient for maintenance purposes.

In the course of analysis, we found it common for transactions that were otherwise dynamically updatable to include minor related changes to add test code, alter initialisation functions, or perform some other cleanup that was not updatable, or was part of another object. These other changes would have been ignored in developing the dynamic update, so we did the same in our analysis and added another series of flags to note when this was done. Of all the dynamically updatable changes, 39% fell into one of these categories. Of only the maintenance updates, we ignored minor parts of 33% of the changes.

The new results in our analysis show that approximately 79% of maintenance changes could directly be converted to dynamic updates. We regard this as a worst-case for our model, because the changes were developed without considering dynamic update, and because some changes to exception handlers might instead have been implemented at a higher level in dynamically-updatable code. We expect that in the maintenance of a real system it would be possible to develop most of the remaining changes as dynamic updates. We will discuss this further in Section 6.2.

5 Evaluation

We conducted experiments to measure the overhead of our update mechanism, and the performance of our system when applying various updates. The results of this evaluation are reported in this section, along with a description of more complex updates enabled by our improvements.

In several of the experiments reported below, we used the ReAIM implementation of the AIM7 multi-user benchmark, in the *alltests* configuration. This benchmark exercises OS services such as IPC mechanisms, file IO, signal delivery, and networking. It was modified slightly to work with K42: we replaced the test using Unix-domain sockets with UDP sockets, altered some code to handle different error return values from K42's Linux emulation library, and prevented the benchmark from removing shared memory regions at the end of its run, because this is not yet supported by K42. We ran the benchmark inside a RAM disk, to avoid IO latencies not imposed by the OS.

All experiments reported here were conducted on an Apple Xserve system, with two 2GHz G5 processors and 512MB of main memory. We built K42 in the no-debug configuration, and ran it in dual-processor mode.

5.1 Costs of mechanism

In this section we examine the added runtime costs of having support for dynamic update in the system. This is much more important than the time to apply an update (which we measure in Section 5.2) because we expect updates to be infrequent events, and because even if the system experiences a slowdown while an update is applied, the advantages over rebooting are significant.

Indirection overhead

First, we consider a property that is part of the fundamental structure of K42: the object translation table's additional indirection on object calls. This indirection adds extra overhead to each object invocation; an object call in K42 requires 6 instructions, instead of 5 for a regular virtual function call. The added instruction is a dependent load, however, because object references are allocated sequentially from a single region of memory, the

object translation table is dense, and thus the extra load is likely to be cached.

It is not possible to directly measure the cost of object indirection, because it is a fundamental part of K42's structure. Instead, we estimated the overhead of object indirection by instrumenting the object system to count the number of indirect object invocations during a run of the ReAIM benchmark. Multiplying this by the number of cycles required for a load from the second-level cache, we estimated the overhead of indirection at less than 0.1% of the total running time on the unmodified system.

Arguably a bigger impact comes from not having a static system structure, preventing application of such cross-module optimisations as inlining, path straightening and dead-code elimination. Furthermore, such a structure replaces direct with indirect branches, and affects the performance of hardware branch prediction. However, any system offering basic module-loader functionality suffers the same problem, and kernel module loaders are now common in commodity operating systems, so this cost has been accepted in those systems.

State tracking overhead

The other added overhead comes from factory objects. During object creation, a factory is now involved, and records the new object's reference. During deletion, the reference is removed from the factory's data structures.

To measure the impact of factories on system performance, we used the ReAIM benchmark described previously. We implemented factories for process objects, one of which exists for each process, and the core memory management objects,[2] a pair of which are created for each open file or mapped memory region. These represent a significant proportion of the objects created in the kernel during benchmark activity: 1,791,808 of all 2,996,920 objects created during a ReAIM run, or 60%, were created using a factory. Hence, we would expect any impact to be visible.

We benchmarked the unmodified system and the modified system with factories added for the above objects. We repeated each experiment three times, taking the average of the peak jobs-per-minute result from each run. We measured a slight (less than 0.5%) performance improvement with the factories present. This is well within the noise caused by changes in code layout and cache behaviour, so we concluded that the use of factories within the kernel has negligible performance impact on the overall system.

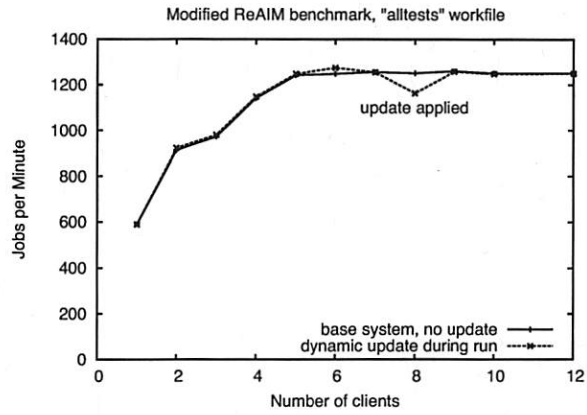


Figure 3: *sync* update applied while benchmarking

5.2 Experiences

Here we describe several example updates enabled by the functionality introduced in Section 4. We also present performance results of applying one of those updates.

File sync patch

When a file is closed, or when its last mapping is removed, the kernel initiates a *sync* operation. This is known as an *unforced sync*, as opposed to the operation that occurs when a program explicitly invokes the *sync* or *fsync* system calls. Because processes block on forced syncs, but unforced syncs only delay the destruction of some buffers, the implementation was changed to prioritise forced syncs over unforced sync. This involved significant changes to the structure of the kernel's *FCMFile* object, with queues of waiting threads and IO operations now maintained differently.

We converted this patch to a dynamic update, by implementing state-transfer functions for *FCMFile* that restructured the internal queues. In total, 53 lines of state transfer code were written.

Figure 3 shows the result of applying this update during a ReAIM benchmarking run. ReAIM incrementally tests higher numbers of clients, so wall-clock time in this graph runs from left to right. We initiated the update when there were eight clients active, and during the next second the overall system throughput dropped, while 170 instances of the *FCMFile* object had their data structures converted on access. Once the benchmark reached the run with nine clients, all affected instances had been transformed, and the update was complete. This can be seen on the graph as a dip in throughput.

While this update was applied, we used a hardware cycle counter to measure the time required for individual phases of the process. Once the update process was started, it took 7ms to instantiate an updated factory ob-

ject and convert the existing factory, and then 487μs to mark all the object instances to be lazily updated. At no point was the whole system's execution blocked.

We also used this update to time our module loader. The module loader is unoptimised, but because the dynamic update process starts once the module has been loaded, this is not of concern. It required 241ms to load the update into the kernel and start executing its initialisation code.

Use of adaptors in large page change

As part of a change made to improve support for multiple page sizes, K42's *root page manager* (PMRoot) class was modified. The *getFrame* and *freeFrame* methods had an argument added specifying the frame size. In this case, there is an obvious default value that can be used for un-updated callers of the interface: the standard page size of 4 kilobytes. Hence, an adaptor object that modifies calls to the PMRoot object can be used. If the method number of a call matches either the *getFrame* or *freeFrame* methods, the relevant argument register is set to 4096.

As part of the overall update for such a change, once the PMRoot object has been updated and an adaptor installed, it is then possible to update the other objects that call PMRoot. This example shows how, in our model, one logical change is implemented as a series of updates, allowing the system to continue making progress.

Interface change to base classes

We provide an example of a dynamic update that is possible with the use of adaptor objects, but has an impact on many objects of different classes due to inheritance. As part of an experiment with new page-allocation policies to avoid fragmentation, a method was added to the *page manager* (PM) class. This class is subclassed by a number of other K42 objects that inherit the new method but also define their own methods. Because a method is added, an adaptor object must be implemented that rewrites the calls made by old callers of the object. Every call made to the old interface with a method number greater than or equal to the newly-added method must have the method number incremented.

The adaptor implementation itself is quite simple, as it only adjusts the virtual function number. However, this example illustrates a scenario where K42's heavy use of C++ and implementation inheritance increases the complexity of updates. Because a method was added to a base class, it was effectively added to all the subclasses, changing their own interfaces, even though the source files did not change. As a result of a single class change, we need to prepare an update for each of the subclasses. A single adaptor implementation suffices, but identifying the affected classes is presently a manual process; we would anticipate a real system automating it, as we will discuss in Section 7.

6 Lessons learned

If we had intended K42 to be dynamically updatable from the beginning, there are several ways in which we would have structured it differently. In this section we discuss solutions to some of the problems we encountered adding dynamic update support to K42.

In our manual analysis described in Sections 3.3 and 4.4, we recorded comments noting why a change couldn't be converted to a dynamic update. The problems generally fell into one of the following categories:

1. changes to static code and data structures, such as low-level code in our exception handlers, general debugging services like our GDB stub functions and in-kernel test system, and system initialisation code;

2. specific services that were developed using static structures and enumerations, primarily K42's tracing service, where each trace point has a unique dense trace identifier, but also glue code used to wrap parts of Linux that run in our kernel;

3. very large cross-cutting changes due to fundamental restructuring of code.

While the third category is probably unavoidable in a research system, it would be unlikely for such changes to occur in the maintenance process of a released system. These include mass changes to interfaces that are not backwards compatible, for example, a number of changes were analysed in which a new argument was added throughout deep call chains across multiple objects. This argument cannot be set by an adaptor, and rewriting the code to support invocation either with or without the argument would be very complex and probably introduce bugs. Similarly, if an interface changes such that the functionality formerly provided by one object is now split across two or more objects, it is very difficult to hide this complexity behind an adaptor and still maintain compatibility for un-updated callers of the interface.

There are however some problems from the other two categories for which we have designed solutions.

6.1 Restructure of initialisation code

The most common example of a static code problem is the kernel initialisation sequence. This code executes once at boot, consists mainly of calls to class-specific

initialisation functions, and cannot be updated. It makes no sense to update the code itself, as once the system has booted any changes will have no effect. However, an update often does need to include initialisation code, for example when introducing a new class into the system, and in many cases this code is the same as the corresponding boot code.

We envision adding a mechanism that allows programmers to annotate initialisation functions in class header files, and automatically calls those functions in a predictable order at boot time, or when the class is loaded as a dynamic update. This is similar to Linux's *initcall* mechanism [29], which uses annotations on functions to be called at boot.

Another related problem is testing code accessed from the kernel console. Currently this is implemented as static functions, and thus is not dynamically updatable. Although test and debugging code is not important for dynamic update in a production system, we would redesign the K42 test system to allow test functions to be registered dynamically, enabling dynamic update as well as dramatically improving the source.

6.2 Exception handlers

Parts of the K42 kernel are not implemented as dynamically-updatable objects, and thus cannot be dynamically updated by our system. This includes low-level exception-handling code, parts of the scheduler, and the implementation of K42's message-passing IPC mechanism. Changes to such code account for the remaining non-updatable maintenance changes in our CVS analysis.

In some cases it is possible to achieve a dynamic update by rewriting other code. For example, on the page-fault path, by implementing a change at a higher level inside the memory management system's dynamically-updatable objects rather than the exception handlers. In other cases, because it is rare for such changes to alter data structures, it may be possible to use indirection available in the exception vectors, or binary rewriting techniques [27] to update the code without the need to achieve quiescence. If data structures were changed, and thus quiescence was required, one could either disable interrupts or run the OS inside a virtual-machine monitor (VMM) [8].

6.3 Dynamic tracing support

K42's tracing service generates a binary log from trace points inserted throughout the system, where each trace point has a unique identifier. To simplify the original implementation, and because we had not considered dynamic update at that time, a static enumeration was used to identify and allocate trace point numbers. To support dynamic updates that add or remove trace points, we could change the static enumeration to a dynamic structure, or use a totally dynamic tracing service [7, 27].

6.4 Conclusion

In general, increased dynamic update coverage can be achieved by minimising statically-bound code, and wherever possible, using structures created at run-time rather than compile-time. This has other advantages: it makes a system more modular, leading to simpler code, better maintainability, and better extensibility.

Our experience shows that when dynamic update is desired, it is usually possible to modify the relevant system structures to enable it. Dynamic update was an addition to K42 after years of development, so it is encouraging that we were able to add it without major structural changes.

7 Future work

7.1 Update preparation

The process of preparing updates for K42 is currently manual. Besides changing the code for a class, state-transfer functions and adaptor objects must be implemented, and programmers must determine the dependencies between updates. Although, as we have shown, it is possible to dynamically update a system using our design, the ideal update system would also automate update preparation from source code changes.

This is not a problem that we are addressing, but it is not unique to operating systems. Other work in the dynamic update field has developed tools to ease the construction of dynamic patches [2, 19] and investigated the semi-automatic creation of state transformers [14, 19], and it would be possible to generate common adaptor objects from source code analysis.

7.2 Reverting updates

In some cases, an administrator may wish to revert or roll back an update after it has been applied. We expect this to occur rarely, since we are considering maintenance updates that have undergone testing by the vendor before their release. Nevertheless, given the data transfer functions and adaptors (which would be developed as part of the update), a *reversal update* that had the effect of reverting to the previous version could be created. However, in any system where updated code runs in the kernel unprotected, if an update has bugs that cause it to corrupt data structures, recovery may be impossible.

7.3 Arbitrary interface changes

As currently designed, our system cannot support interface changes in which the changes cannot be hidden behind an adaptor. While this is not a problem for all but the most substantial new features and code restructuring changes, for completeness we would like to support all dynamic updates.

As explained previously, support for arbitrary interface changes requires blocking all objects affected by a change, and potentially the whole kernel. We would block kernel events at a lower level, such as exception handlers or underlying VMM, before updating the system. Although this precludes servicing requests while the update is applied, it preserves the system's full state, and thus offers significant advantages over rebooting.

7.4 Updates outside the kernel

Many OS updates change user-level code such as system libraries. Although we have targeted kernel changes, a complete dynamic update system would also require support for user-level updates. There is nothing preventing our update mechanism from operating at user-level. However, depending upon the structure of the relevant libraries or applications, general-purpose dynamic update systems [2, 19, 24] may be more suitable or practical.

7.5 Implementation in other systems

As our approach has been to implement dynamic update within an existing OS with the aim of developing a design suitable for commodity operating systems, one goal for future work would be to apply it to a commodity OS. In previous work [5], we described how dynamic update may be implemented in Linux. Although it does not provide the same consistent mechanisms, modular parts of Linux such as the VFS layer and device driver interfaces are effectively object-oriented, providing data hiding and indirection. For example, filesystem drivers are invoked through a table of function pointers held in the *inode* structures, and device drivers are called similarly. This provides the same indirection as K42's object translation table, although at a coarser granularity. It would result in a lower overhead for indirection and state tracking, but potentially higher update costs. The design's other requirements, such as state tracking, can also be achieved within Linux, albeit not so consistently as in K42 [5].

One significant difference between K42 and Linux is blocking threads. K42 kernel threads are short-lived and non-blocking, allowing us to block access to an object and simply wait until existing threads terminate to achieve quiescence. In Linux, however, system call handlers may block for IO or other long-running operations, that we cannot wait for. Our current solution when applying an update in Linux is, when possible, to abort system calls with *EINTR* (interrupted call) or *EAGAIN* (resource temporarily unavailable) errors, from which an application can recover by retrying the call. If a blocking call cannot be interrupted, we must delay and retry the update until it completes. This could be avoided by a wrapper that converts blocking system calls into restartable variants such as *select*.

The main limitation in applying this approach to Linux is the current extent of modularity. Unlike K42, core parts of Linux such as the scheduler and virtual memory system are not modular. Despite this, we believe that dynamic update for Linux is feasible. In particular, it is not necessary to apply modularisation and dynamic update infrastructure throughout the kernel. Rather, dynamic update can be enabled incrementally for specific subsystems, by adding indirection and state tracking (or subverting existing structures). Along with other projects [26, 28], this provides motivation for increasing the modularity of the Linux kernel.

8 Related work

Many dynamic update systems exist for high-level languages [3, 10, 14, 20, 25], however these are inapplicable to an OS implemented in C or C++. A small number of general-purpose dynamic update systems for C have been described in the literature [2, 12, 19]. These focus on application code, however an OS kernel is a fundamentally different environment, and features constraints such as a high level of concurrency, and completely event-driven execution and control flow. Also, operating systems offer extremely limited runtime environments. These constraints result in different trade-offs and a different design for dynamic update.

Most general-purpose dynamic update systems for C do not support threading [12, 19]. One that does is OPUS [2], that uses Linux's *ptrace* facility to update C programs at function boundaries, with the goal of enabling dynamic security patches. OPUS waits for updated functions to be off the stack of all threads. Unlike our system, which blocks new invocations to achieve quiescence, it is possible for OPUS never to achieve quiescence, and thus for an update to be delayed indefinitely. Because OPUS relies on stopping all threads to examine their stacks, it would be difficult to apply the design to an operating system kernel, where thousands of threads may be present, and where blocking the whole system's execution is not feasible. OPUS does not handle changes to data structures nor function interfaces. Despite these limitations, it was able to apply many real security patches, suggesting that our system is also suitable for security patches.

LUCOS [8] is a dynamic update system for Linux built

upon the Xen virtual-machine monitor. To apply updates, LUCOS enforces quiescence by using the VMM to stop the system's execution; it then dynamically patches functions. In contrast, in K42 we block and quiesce only objects affected by an update, allowing the rest of the system to continue. Similarly to OPUS, LUCOS does not support changes to function interfaces, that are important even for bug fixes, as we have shown. If a LUCOS patch changes data structures, pages containing old and new versions of the affected data are marked read-only; on a write fault the kernel is single-stepped, and a data-transfer function is used to keep the versions consistent. K42 avoids such two-way data conversions by updating all code that accesses a data structure along with the data itself. For LUCOS to detect when functions using old data structures have returned, it must examine every stack frame of every kernel thread in the system, a large scalability problem on modern systems that commonly run thousands of threads. Because it only consists of passive modules, LUCOS' performance overhead is negligible at the expense of higher update costs, especially when data structures are changed. Our system incurs constant overhead from indirection and state tracking, although, as we have shown, this overhead is very low (less than 1%). Furthermore, our system's overhead compares favourably with virtualisation, and enables a simpler and more scalable update process.

DynAMOS [15] is another recent dynamic update system for Linux. Like LUCOS, it uses function-level binary patching, which occurs while interrupts are disabled. DynAMOS supports limited updates to data structures through the use of shadow structures, that must be maintained separately to the original structure. Like LUCOS, DynAMOS may need to walk the stack of every kernel thread to detect quiescence. As a loadable module that requires no modifications to the base kernel, DynAMOS does not impace base performance, however because every updated function incurs an extra indirect branch, the performance impact of updates is significant. Micro-benchmark results show overheads higher than 40% for some functions, but the overall performance impact of an update is not reported.

The main advantages of LUCOS and DynAMOS over our approach are that they do not require changes to the kernel, are able to update almost any function, and do not incur any base performance impact. However, due to the use of binary function patching as the underlying mechanism, the impact of applying updates in these systems is significantly higher. Although our approach has not yet been applied to Linux, the comparison between it and these systems is primarily a trade-off between applicability, in terms of the parts of the kernel that can be updated, and performance and complexity, in terms of the cost of applying updates, the difficulty in developing them, and the complexity of changes that are supported.

A practical approach to general-purpose dynamic update is taken by Ginseng [19]. Ginseng compiles C programs specially, adding indirections for types and functions, to provide safe, fine-grained dynamic updates for arbitrary C code. This contrasts to our approach, that applies updates at the coarser level of objects or modules, and so suffers lower overheads from indirection and update support, but relies on the modular structure of the OS. Ginseng does not support threaded execution, although the authors are currently investigating ways to cope with the concurrency requirements of an OS [18].

Several domain-specific approaches to dynamic update appear in the literature, for example in object databases [6] and distributed systems [1]. These systems are not necessarily language-based, but are closely tied to their domain for other reasons. Our approach is similar in that we have tailored the design of our system to the structure and environment of the code we update, namely modular kernel code.

One relevant domain-specific dynamic update system is Upstart [1], which provides automatic software updates for distributed systems by interposing at the library level and rewriting remote procedure calls. Despite the different focus, a lot of parallels can be drawn between Upstart and K42. Upstart's transform functions are equivalent to our state-transfer functions, simulation objects play the same role as adaptors, and scheduling objects are a generalisation of lazy update.

AutoPod [22] is a kernel module for Linux that provides checkpoint, migration and restart of processes transparently to the applications and kernel. Combining AutoPod with an underlying VMM, as long as the kernel interface is unchanged it is possible to start a new virtual machine with an updated kernel, checkpoint the user processes in the old virtual machine, and migrate and then restart them on the updated kernel. This represents a radically different approach to the problem; it avoids significant changes to the OS, at the cost of runtime overhead from virtualisation and AutoPod.

9 Conclusions

We have implemented dynamic update for K42, and found that adding this feature to an existing operating system is feasible when that system has a sufficiently modular structure. K42 has the advantage that it is object oriented with a consistent and pervasive module invocation mechanism, but as we have discussed, commodity operating systems such as Linux can provide the same support, albeit in an less consistent fashion.

We have shown how to design a dynamic update system that handles interface changes, which are required for many bug fixes, and applies updates lazily, which is

essential for changes to some data structures because it mitigates their severe performance impact.

From a study of the K42 revision history, we have shown that our dynamic update model can support at least 79% of maintenance changes to an operating system, and we expect that for a real system the proportion would be closer to 100%. We have also measured the performance impact, and found it to be insignificant, with less than 1% runtime overhead.

Dynamic update is a rich area for future systems research, and will become increasingly important for mainstream operating systems. We have shown one way to achieve a dynamic update feature in operating systems. With dynamic update, the uptime of systems should be limited by hardware failure, not by software updates.

Acknowledgements

The K42 community provided valuable assistance—Raymond Fingas developed the *arbiter objects* used for lazy update and adaptors, and Jeremy Kerr implemented the *sync* changes. Comments and critical feedback were provided by the participants of the First EuroSys Authoring Workshop, Eno Thereska, Iulian Neamtiu, and the anonymous reviewers.

This work was partially supported by DARPA under contract NBCH30390004. National ICT Australia is funded by the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

Notes

[1]We examined the *kitch-core* and *kitch-linux* CVS modules.

[2]In K42 terminology, we are referring to the file cache manager (FCM) and file representative (FR) objects.

References

[1] S. Ajmani. *Automatic Software Upgrades for Distributed Systems*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Sep 2004. Also as Technical Report MIT-LCS-TR-1012.

[2] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online patches and updates for security. In *14th USENIX Security Symp.*, pages 287–302, Aug 2005.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*, chapter 9, pages 121–123. Prentice Hall, 2nd edition, 1996.

[4] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *2005 Ann. USENIX*, pages 279–291, Apr 2005.

[5] A. Baumann, J. Kerr, J. Appavoo, D. Da Silva, O. Krieger, and R. W. Wisniewski. Module hot-swapping for dynamic update and reconfiguration in K42. In *6th Linux.Conf.Au*, Apr 2005.

[6] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA*, pages 403–417, Oct 2003.

[7] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *2004 Ann. USENIX*, pages 15–28, Jun 2004.

[8] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew. Live updating operating systems using virtualization. In *2nd VEE*, pages 35–44, Jun 2006.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[10] S. Gilmore, D. Kírlí, and C. Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, Department of Computer Science, The University of Edinburgh, Dec 1997.

[11] P. Gray. Experts question Windows patch policy. ZDNet News, Nov 2003. http://news.zdnet.com/2100-1009_22-5105454.html.

[12] D. Gupta and P. Jalote. On-line software version change using state transfer between processes. *Softw.: Pract. & Exp.*, 23(9):949–964, Sep 1993.

[13] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *EuroSys Conf.*, pages 133–145, Apr 2006.

[14] I. Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, University of Wisconsin-Madison, May 1983.

[15] K. Makris and K. D. Ryu. Dynamic and adaptive updates of nonquiescent subsystems in commodity operating system kernels. In *EuroSys Conf.*, Mar 2007.

[16] K. B. Mierle, K. Laven, S. T. Roweis, and G. V. Wilson. CVS data extraction and analysis: A case study. Technical Report UTML TR 2004-002, Dept of Computer Science, University of Toronto, Sep 2004.

[17] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *2nd MSR*, pages 2–6, May 2005.

[18] I. Neamtiu and M. Hicks. Dynamic software updating for the Linux kernel. OSDI, Work-in-Progress Session, Nov 2006. Seattle, WA, USA.

[19] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI*, Jun 2006.

[20] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Int. Conf. Softw. Maintenance*, Oct 2002.

[21] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in Linux device drivers. In *EuroSys Conf.*, pages 59–71, Apr 2006.

[22] S. Potter and J. Nieh. Reducing downtime due to system maintenance and upgrades. In *19th LISA*, pages 47–62, Dec 2005.

[23] E. Rescorla. Security holes... who cares? In *12th USENIX Security Symp.*, pages 75–90, Aug 2003.

[24] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *Softw.*, 10(2):53–65, Mar 1993.

[25] D. Stewart and M. M. T. Chakravarty. Dynamic applications from the ground up. In *ACM SIGPLAN Haskell WS*, Sep 2005.

[26] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *6th OSDI*, Dec 2004.

[27] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *3rd OSDI*, pages 117–130, Feb 1999.

[28] E. Witchel, J. Rhee, and K. Asanović. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *20th SOSP*, Oct 2005.

[29] T. Woerner. Understanding the Linux kernel initcall mechanism, Oct 2006. http://geek.vtnet.ca/doc/initcall/.

[30] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *1st MSR*, May 2004.

Exploring Recovery from Operating System Lockups

Francis M. David, Jeffrey C. Carlyle, Roy H. Campbell
University of Illinois at Urbana-Champaign
{*fdavid,jcarlyle,rhc*}*@uiuc.edu*

Abstract

Operating system lockup errors can render a computer unusable by preventing the execution other programs. Watchdog timers can be used to recover from a lockup by resetting the processor and rebooting the system when a lockup is detected. This results in a loss of unsaved data in running programs. Based on the observation that volatile memory is not affected when a processor a reset occurs, we present an approach to recover from a watchdog reset with minimal or zero loss of application state. We study the resolution of lockup conditions using thread termination and using exception dispatch. Thread termination can still result in a usable system and is already used as a recovery strategy for other errors in Linux. Using exceptions allows developers to write code to handle a lockup within the erroneous thread and attempt application transparent recovery. Fault injection experiments show that a significant percentage of lockups can be recovered by thread termination. Exception handling further improves the recoverability of the operating system.

1 Introduction

While many techniques have been invented over the years to create software that is resilient to faults [1], errors due to hardware and software faults still remain a serious problem in today's world. Some errors can cause the processor to lock up in an infinite loop of useless computation. In this case, the error can only be detected by an external entity. Lockup errors that happen in user programs can be detected by other programs [2] and can usually be handled without affecting unrelated programs. On the other hand, lockups that occur inside the operating system (OS) can render the computer unusable by not allowing any other programs to execute. Lockup causing bugs are common in OS code. More than 30% of the bugs in Linux discovered by Chou et al. [3] were bugs that could potentially cause a lockup.

Watchdog timers have been traditionally used to detect lockups in OS code and are usually configured to reset the processor when the timer expires. To prevent a processor reset and a consequent reboot, the OS must periodically reset the watchdog timer. It is common to refer to watchdog expiration as a bite and to the act of resetting the watchdog as a kick. While rebooting after a lockup improves availability, it results in a loss of all running user programs and data.

In this paper, we demonstrate that this reboot behavior can be replaced by an approach where the reset signal to the processor is used to recover the system with minimal or zero loss of application state. The key observation that motivates our approach to OS recovery is that the reset signal only affects the processor and leaves volatile memory intact. Information loss is limited to the contents of the processor at the time of the reset and the contents of volatile memory can be used for recovery.

We have implemented watchdog based recovery in Linux and in the Choices object-oriented OS [4]. We explore recovery after a watchdog bite using two methods: by terminating the locked up thread (in Linux and Choices), and by dispatching a C++ exception to the thread (in Choices).

Using thread termination is a simple approach to recovery. There is no attempt to prevent or fix possible kernel data structure inconsistencies. Therefore, there are no guarantees that the system is successfully recovered. However, attempting to recover a crashed system by terminating a thread is not uncommon. Operating systems such as Linux already respond to kernel space errors like invalid pointer dereferencing by terminating the erroneous thread. We, therefore, apply this approach to lockup errors as well. Experiments with Choices and with Linux demonstrate that recovery is possible from a wide variety of OS lockups when using thread termi-

Part of this research was made possible by grants from DoCoMo Labs USA and generous support from Texas Instruments.

nation. We consider a recovery attempt to be successful if the OS continues to schedule and run other existing threads and provides some minimal functionality like filesystem and console access.

In Choices, we are exploring the use of the C++ exception mechanism as a unified framework for notification of all errors that occur within the OS. We create exceptions from errors like memory faults, invalid instructions and hardware aborts and allow threads to respond to them using exception handlers [5]. We were therefore motivated to add OS lockups to the set of exceptions already handled by the Choices kernel.

Existing techniques such as Nooks [6] and SafeDrive [7] do not attempt to recover from lockup errors within extensions. Our watchdog timer based recovery approach complements both these techniques and enables them to recover from a larger class of errors. OKE [8] can detect and recover lockup errors in extensions compiled with a safety enforcing trusted C compiler. Our recovery approach does not require a special compiler and works with existing code. Also, these systems only consider errors in device drivers. Thread termination and exception dispatch can be used to recover from lockup errors in other parts of the OS as well. For example, a lockup in a non-preemptable system call handler is not detected by Nooks, SafeDrive, or OKE; but is detected and potentially recovered by our techniques.

Our recovery implementations have been evaluated on two ARM processor based platforms: the Texas Instruments OMAP1610 H2 hardware development kit and the QEMU [9] system emulator.

While the traditional action of a watchdog has been to reset the system on a watchdog bite, an alternative would be to raise a non-maskable interrupt (NMI). Current ARM processors do not support non-maskable interrupts. Nevertheless, we still examine the advantages of using a non-maskable interrupt to notify the processor of a watchdog bite.

2 Watchdog Recovery Design

A careful analysis of OS behavior is required when deciding when to execute watchdog kicks. Placing the kick code in the timer interrupt handler ensures that, as long as interrupts are enabled, and there is no lockup in the interrupt handler, the watchdog will not bite. However, it is possible that a lockup can occur with interrupts still enabled. If the lockup is in a non-preemptable section of code, the OS is unusable because it does not schedule any other threads. In Linux, watchdog timers are exported as devices to userspace and the kicks are issued periodically by a userspace thread. If the userspace thread does not get scheduled because of an OS lockup, the watchdog reboots the system. Kicks issued from threads are a more effective indication of the system being alive than kicks issued from timer interrupts.

When the ARM processor is reset, it switches to a privileged execution mode and sets its program counter (PC) to address 0. The signal also resets the interrupt controller and all interrupts are turned off. The memory management unit (MMU) is also turned off and only physical addressing is possible. Address 0 is normally the start address of the bootloader. The bootloader's job is to initialize the memory hardware and load the OS kernel into RAM from flash memory or secondary storage. It then relinquishes control to the OS. The bootloader usually does not differentiate between resets attributed to watchdog timers and power-on resets. Thus, the OS is always reloaded and rebooted, causing a loss of all running programs and data in memory.

In order to ensure that memory contents are preserved, the bootloader needs be modified to treat the watchdog bite differently. When the watchdog bites, the bootloader should not reload the kernel and should instead directly transfer control to the OS start address in memory. This is a reasonable approach because, once it is up and running, the OS core is never paged out and resides in the same physical memory area into which it was first loaded.

Once control is back in the OS, a recovery routine can take over. The recovery process involves switching the MMU back on, initializing the interrupt controller, re-enabling interrupts and performing an appropriate action to eliminate the lockup condition before starting to schedule threads again.

There are a couple of issues that arise when attempting to recover from watchdog bites that reset the processor. This requires that the processor cache is configured as write-through instead of write-back in order to avoid loss of cached data. Thus, when using this technique, we gain increased reliability at the expense of some decreased performance. Also, part of the processor context at the time the watchdog bites is lost forever. For example, the program counter is instantaneously overwritten by 0. This makes it difficult to accurately pinpoint the location of the lockup and debug the error. Both these issues cease to exist if the watchdog timer is wired to a non-maskable interrupt. This would enable the OS to respond to the lockup without any loss of information in the processor or the cache. Additionally, using an NMI simplifies the recovery implementation because the MMU and interrupt controllers are not disturbed.

3 Recovering Linux

Soft Lockup Detector: A soft lockup is an error condition where a thread is locked up in kernel mode with

interrupts enabled. Some soft lockups can render the system unusable by permanently preempting all other threads. The Linux kernel includes code that detects these kinds of soft lockup errors. A low priority thread updates a timestamp every second. This timestamp is checked during a timer interrupt to see if it was updated within the last ten seconds. This ensures that the system is usable by confirming that the watchdog thread is periodically scheduled. If the check fails, the detector displays a message reporting the lockup error and records it in the system logs. The detector does not attempt to fix the error.

In order to study the recoverability of the Linux kernel from a lockup detected by the soft lockup detector, we added code to terminate the thread which has locked up in kernel mode. Linux already handles most errors that are encountered within the kernel by terminating the thread. These are usually called "Oops" errors. However, if an "Oops" occurs in interrupt mode, the error is deemed to be serious and the "Oops" handler calls `panic()` which halts the system. Kernel code can also directly call `panic()` on detecting a serious error. We do not attempt to recover from Linux kernel panics.

The soft lockup detector cannot detect lockups that occur when interrupts are disabled because the detector code is not executed. These "hard" lockups can only be detected using an external hardware watchdog timer.

Hardware Watchdog: We added a new kernel thread that wakes up periodically and kicks the watchdog timer. If this thread is not scheduled periodically, the processor is reset. A normal power on reset causes the bootloader to load a compressed kernel image into RAM and transfer control to the header in the compressed image. The header then runs a decompression routine which places the kernel at some platform dependent physical address. The kernel is always resident at this physical address. We modified the small bootloader built into QEMU so that it does not reload Linux and instead directly jumps to the start address of the existing uncompressed kernel when a reset is generated by the watchdog timer.

We modified the first few instructions in the Linux boot up code to check for the reset reason. If the reset was due to the watchdog timer, a recovery routine is executed. The MMU is turned on first with the page tables configured for kernel tasks. Switching on virtual memory ensures that all kernel data structures are visible again. The task that was running at the time of the watchdog bite is then terminated. In the next stage, peripheral interrupts and the watchdog timer are re-enabled. The code then enters the processor idle loop which works as a dispatcher for runnable threads. Recovery is completed once the idle loop begins picking up runnable threads and scheduling them on the processor.

We do not need to worry about locks held by the thread when it is terminated because our target platform is a uniprocessor. Linux implements spin locks on uniprocessors by disabling interrupts for the duration that the lock is held. Thus, if a thread locks up when holding a spin lock, it can only be detected by a watchdog timer. The lock is implicitly released after recovering from a watchdog bite. On multi-processor hardware, lock usage tracking functionality is required in order to release all locks held by the thread when it is terminated. This can be implemented easily by modifying the spin lock functions or by using a code rewriting approach [7]. Usage of semaphores in the locked up thread can present some problems with recovery. We expect tracking semaphore usage to improve chances of successful recovery, but we have not yet explored this direction.

It is also possible that kernel data structures are left in an inconsistent state after a thread is abruptly terminated. This might be unacceptable in high integrity systems. Data structure usage tracking techniques such as those used in Nooks can help mitigate this issue. Fixing or preventing kernel data structure corruption is itself a significant challenge and we do not address it in this work.

These issues with locking, semaphore usage and data structure corruption are identical to those that occur when Linux encounters "Oops" errors. The default response in Linux is to terminate the thread without worrying about any of these issues. Thus, lock and semaphore tracking can also improve recoverability in this case.

As described in section 2, the use of an NMI allows for improved performance and improved debugging support. Some recent x86 interrupt controllers can be configured to generate periodic non-maskable interrupts to the processor. The x86 version of the Linux kernel includes support for lockup detection which exploits this functionality. Similar to the soft lockup detector, the NMI driven detector displays an error message when it detects a lockup. This support is however not yet available on the ARM platform.

4 Recovering Choices

Hardware Watchdog: Recovery from a processor reset issued by a hardware watchdog has also been implemented in Choices. Choices does not yet support soft lockup detection. Soft lockups do not result in an unusable system because the kernel is fully preemptable.

The watchdog is kicked at every timer interrupt. If timer interrupts are not received because of a hard kernel lockup, the watchdog bites. Just as with Linux, Choices invokes a recovery routine instead of proceeding with normal boot if the reset reason was a watchdog timeout.

The recovery routine pretends to be the idle thread and switches the MMU on and restores interrupts. It then pretends to be the locked up thread and calls `die()` directly.

The recovery procedure differs from our Linux implementation. In Linux, we restored the idle thread and it picks up and kills the locked thread. In Choices, we directly kill the locked thread and this automatically restores the next runnable thread on the processor. Both these approaches are valid and either one can be chosen depending on ease of implementation.

While a thread termination approach might help the kernel to continue scheduling other threads, it might still render the OS unusable because the terminated thread might be a critical kernel thread. We have previously explored the use of C++ exceptions to notify threads of errors they encounter in kernel space [5]. Using exceptions allows threads to attempt local recovery strategies in exception handlers. We were therefore motivated to explore converting a thread lockup condition into a C++ exception.

An exception can only be properly dispatched by the C++ exception handling libraries if the context in which the exception is thrown is correct. Thus, simply writing a C++ throw statement in the recovery routine will not work. We needed a way to recover the context of the locked up thread at the time of the watchdog bite. After some experimentation, we discovered that the processor does not lose the contents of most of its registers when it is reset. The PC is lost because it is reset to 0x0, and the value of the processor status register is also lost. But the contents of all the other registers are preserved.

We modified the bootloader to respond to a watchdog bite by storing the contents of the reset preserved registers before they are clobbered by running the recovery routine. A valid value of PC needs to be recovered for exception dispatch to work. We choose to approximate the value of the PC as the first instruction of the function in which the lockup occurred. In machine code generated by the GNU C++ compiler, the PC is saved on the stack frame in the preamble of every function. We can read the last saved PC from the stack using the recovered stack frame pointer register and use this value. The context is now usable for dispatching an exception. This context is modified so that when it is restored on the processor, it enters a helper function which uses the C++ throw keyword to raise an exception.

Standard C++ try-catch syntax can be used to handle these exceptions. We believe that this is an elegant approach to handling lockup conditions within an erroneous thread in kernel mode. A developer can write an exception handler to try thread-specific recovery strategies if the thread ever locked up. Also, unlike the thread termination approach, lockup exception handling can be used within kernel contexts like the initial interrupt processing code which is not a part of any thread.

Table 1: Lockup detection and recovery for Non-Preemptable and Preemptable Linux (*-Y with enhancement)

Lockup Location	Software				Watchdog			
	Det?		Rec?		Det?		Rec?	
Interruptible thread	Y	N*	Y	N*	Y	N	Y	N
Non-interruptible thread	N	N	N	N	Y	Y	Y	Y
Interrupt handler	Y	Y	N	N	Y	Y	N	N
Syscall handler	Y	N*	Y	N*	Y	N	Y	N

5 Evaluation

Linux: The 2.6 series of kernels have experimental support for kernel-mode preemption and this affects lockup detection. We, therefore, evaluate our implementations with both a non-preemptable and a preemptable kernel.

We introduced artificial infinite loop bugs into different types of kernel contexts and studied the detection and recovery properties of the kernel software detector and a hardware watchdog with kicks issued by a kernel thread. The thread that is terminated for recovery is a non-critical dummy thread and there is no memory corruption.

Table 1 catalogs our experiences with both a non-preemptable and a preemptable version of the kernel. As expected, the soft lockup detector is unable to detect lockups when interrupts are disabled. In these cases, the watchdog timer is able to detect and recover the system. Linux allows nested interrupts and therefore interrupts are enabled when running an interrupt service routine (ISR). A lockup in an ISR, which is non preemptable, is therefore detectable by the soft lockup detector. Recovery is not possible because Linux does not support termination of a thread executing in interrupt context.

Lockup detection effectiveness is reduced when experimental kernel mode preemption support is turned on. This ensures that the watchdog thread is always scheduled even when a higher priority preemptable thread enters a lockup in kernel mode. This is an unfavorable situation because, even though the system is usable, the locked up thread keeps the processor busy. It is possible to detect such situations by measuring the time spent by a thread in kernel space without yielding. A kernel developer has posted a patch for the x86 architecture that enables the soft lockup code to detect these lockups[1], but this has not yet been included in the mainstream kernel. In the table, entries marked with an asterisk can be changed to "Y" with such an enhancement.

[1] `http://lkml.org/lkml/2005/8/2/216`

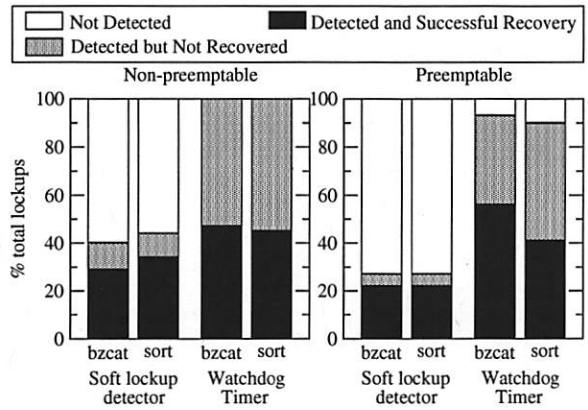


Figure 1: Linux lockup detection and recovery efficiency

When kernel preemption is turned on, the existing soft lockup detector can still detect lockups that occur during a period when preemption is temporarily disabled, resulting in the system being unable to schedule other threads. For example, lockups in interrupt handlers can be detected because these cannot be preempted.

We also performed automated lockup fault injection experiments into various parts of the kernel using a modified QEMU. We randomly pick instruction addresses into which faults are to be injected. A fault is injected by changing the chosen instruction to a self-loop. The fault is transient and is not re-encountered if the instruction is executed again. We inject only one lockup in each experiment. In one set of experiments, faults are injected when running a bzip2 decompression task (bzcat). In another set, faults are injected when running a sort task. Our goal is to examine if lockups in random parts of the kernel affect the successful completion of these user tasks.

We measure the number of lockup detections (using both the soft lockup detector and the watchdog timer) and the number of successful completions of these tasks after recovery is attempted. A successful completion is defined as a run that produces output identical to a run without fault injection. In all our experiments, there are several running background tasks; some of which are standard Linux kernel threads.

The results of our experiments are shown in figure 1. For the non-preemptable version of the kernel, the soft lockup detector detects less than 40% of lockups because most of them occur when interrupts are disabled. The system does not recover when the lockup goes undetected. The watchdog timer detects all the lockups. But, in spite of this increased detection efficiency, the user task only completes correctly in around 50% of the lockups. The reasons for unsuccessful recovery (after detection) vary. Our analysis reveals that between 80-100% of these were because the detection occurred when the kernel was in interrupt context. Since the kernel calls `panic()` when a thread is terminated in interrupt context, the system does not recover.

In the preemptable kernel experiments, there are several lockups that do not cause a complete system crash because they are preempted (not shown in the figure). These are not detected by either the soft lockup detector or the watchdog timer. In 7-10% of the lockups, the tasks complete successfully in spite of the lockup not being detected. These represent the cases in table 1 marked with an asterisk.

For the preemptable kernel, the watchdog has an edge over the soft lockup detector because it can detect lockups when interrupts are disabled.

Choices: The Choices kernel is designed to be preemptable and watchdog timers are only used to detect hard lockups. In order to test our watchdog recovery implementation in Choices, we first inserted artificial lockup bugs into a dummy kernel thread in Choices. Choices is able to recover from bugs in interruptible, non-interruptible and system call handlers by terminating the dummy thread. Unlike Linux, the design of Choices allows it to be recovered from lockups in interrupt handlers as well. Dummy threads and transient lockups were also used to test correct operation of the lockup exception dispatch mechanism in Choices.

We also performed fault injection experiments with Choices in a manner similar to that described for the Linux experiments. We use user tasks represented by a sort program and a gunzip decompression program. Hard lockup errors (infinite loop with interrupts disabled) are injected into the Choices kernel. The watchdog detects all the hard lockups errors that are encountered. Experiments are performed for both the thread termination and the exception handling approaches to recovery. C++ "catch" statements are used in several top level objects to handle exceptions by retrying the request.

Figure 2 compares the recovery capabilities of the

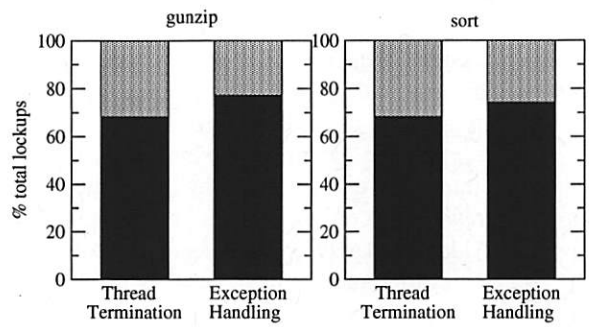


Figure 2: Choices hard lockup recovery comparison

exception based approach with the thread termination based approach. Handling errors using exceptions results in the user task completing successfully from about 6-9% more lockups than when using thread termination. This is because exception handlers in various OS objects attempt to recover the locked thread by retrying the method call that failed and some of these retries are successful. Non-recovered lockups are mostly due to data structures left in an inconsistent state. These results are based on only a few exception handlers in top level objects. We expect that a more thorough deployment of exception handlers throughout the object hierarchies will reduce data structure inconsistency issues that prevent recovery and result in improved recoverability.

6 Discussion and Related Work

In addition to watchdog timers, lockups can also be detected by hardware such as the RSE [10]. RMK [11] detects an OS lockup by counting the number of instructions between two consecutive context switches.

Our approaches to recovery deviate from a fail-stop model of computation in a manner similar to failure oblivious computing [12]. A full system reboot may have to be performed in order to completely recover the system. Our approaches can be combined with techniques like isolation containers [13, 6], microreboots [14] and data structure repair [15] to improve recoverability.

There is some directly related research in application recovery after OS crashes. The recovery box approach [16] uses non-volatile memory to store application state that is restored when the system is restarted after a crash. Remote-DMA can be used to access the memory of a crashed system and recover application state [17]. In the Rio filesystem [18], the buffer cache is recovered from volatile memory after a reset. In contrast to these approaches, we attempt to recover the entire system.

7 Summary and Conclusions

We have discussed detection of operating system lockup errors using software detectors and watchdog timers. We have shown that it is possible to recover from a significant number of such lockup errors by terminating the locked up thread. We have also shown that the use of simplistic retry based exception handling to attempt recovery provides up to a 9% improvement in recoverability. We expect this number to increase with increased deployment of exception handling within the OS.

Additional details and code are available online at `http://choices.cs.uiuc.edu/`.

References

[1] Wilfredo Torres-Pomales. Software Fault Tolerance: A Tutorial. Technical Report NASA/TM-2000-210616, NASA, 2000.

[2] Keith Whisnant, Ravishankar K. Iyer, Zbigniew T. Kalbarczyk, Phillip H. Jones III, David A. Rennels, and Raphael Some. The Effects of an ARMOR-Based SIFT Environment on the Performance and Dependability of User Applications. *IEEE Trans. Softw. Eng.*, 30(4):257–277, 2004.

[3] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An Empirical Study of Operating System Errors. In *SOSP*, pages 73–88, 2001.

[4] R. H. Campbell, G. M. Johnston, and V. Russo. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)". *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[5] Francis M. David, Jeffrey C. Carlyle, Ellick M. Chan, David K. Raila, and Roy H. Campbell. *Exception Handling in the Choices Operating System*, volume 4119 of *LNCS*. Springer-Verlag Inc., 2006.

[6] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering Device Drivers. In *OSDI*, pages 1–16, 2004.

[7] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harre, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *OSDI*, Nov 2006.

[8] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *IEEE Open Architectures and Network Programming*, 2002.

[9] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX ATC, FREENIX Track*, April 2005.

[10] Nithin Nakka, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Jun Xu. An Architectural Framework for Providing Reliability and Security Support. In *DSN*, pages 585–594. IEEE Computer Society, 2004.

[11] Long Wang, Zbigniew Kalbarczyk, Weining Gu, and Ravishankar K. Iyer. An OS-level Framework for Providing Application-Aware Reliability. In *12th IEEE Pacific Rim International Symposium on Dependable Computing*, 2006.

[12] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr William S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI*, pages 303–316, 2004.

[13] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP*, pages 207–222, New York, NY, USA, 2003. ACM Press.

[14] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *OSDI*, San Francisco, CA, December 2004.

[15] B. Demsky and M. Rinard. Automatic Data Structure Repair for Self-Healing Systems. In *Proceedings of the First Workshop on Algorithms and Architectures for Self-Managed Systems*, San Diego, California, June 2003.

[16] Mary Baker and Mark Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. In *USENIX*, pages 31–44, Summer 1992.

[17] Florin Sultan, Aniruddha Bohra, Stephen Smaldone, Yufei Pan, Pascal Gallard, Iulian Neamtiu, and Liviu Iftode. Recovering Internet Service Sessions from Operating System Failures. *IEEE Internet Computing*, 9(2):17–27, 2005.

[18] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Architectural Support for Programming Languages and Operating Systems*, pages 74–83, 1996.

Supporting Multiple OSes with OS Switching

Jun Sun[1], Dong Zhou, Steve Longerbeam[2]
zhou@docomolabs-usa.com
DoCoMo USA Labs
3240 Hillview Ave., Palo Alto, CA 94304, USA

Abstract—*People increasingly put more than one OSes into their computers and devices like mobile phones. Multi-boot and virtualization are two common technologies for this purpose. In this paper we promote a new approach called OS switching. With OS switching, multiple OSes time-share the same computer cooperatively. A typical implementation can reuse an OS's suspend/resume functionality with little modification. The OS switching approach promises fast native execution speed with shorter switching time than traditional multi-boot approach. We describe the design of OS switching as well as our implementation with Linux and WinCE, and evaluate its performance.*

1. Introduction

Many people nowadays run multiple OSes on their computers. For example, developers may need to test their software on different OSes and/or on different versions of the same OS. Users sometimes find that two pieces of software that they like require different OSes.

In mobile device paradigm, we are also seeing more and more applications/systems using multiple OSes. For example, VirtualLogix (formerly Jaluna) [1] provides a solution for phones that integrates a real-time communication OS with a Linux OS. The recent announcement of OSTI [2] also indicated a trend of having multiple OSes on one mobile device.

Up until now there are two main approaches to supporting multiple OSes: multi-boot and virtualization [3]. With multi-boot systems, a user installs multiple OSes into disjoint disk partitions along with a multi-OS-aware boot loader. During the boot up time, the boot loader asks the user to select the OS to boot [4]. To run an OS different from the current OS, the user exits the current OS and reboot into the other OS through the boot loader. While switching to another OS takes a long time, the approach has the advantage of each OS running directly on hardware without any modification and running with full speed and full access to hardware resources.

Virtualization technology has been very popular recently (see [5][6][7][8] as examples). In virtualization, a user typically installs Virtual Machine (VM) monitor and related management software. With the help of VM software the user can further install several different OSes onto the same computer. The VM software can typically run multiple OSes concurrently. For example, with VMWare Workstation, each OS has its display shown as a window on the host OS. Switching from one running OS to another is almost equivalent to switching between GUI applications. However, virtualization technology typically suffers from degradation in performance [5]. More importantly, it typically requires a considerable amount of work to providing a virtualizing and monitoring layer, and sometimes to modify the existing OS and its device driver code.

In this paper we promote an alternative approach, called *OS switching*. In OS switching, multiple OSes time-share the same computer cooperatively. A straightforward implementation of OS switching can make use of the suspend/resume features that already exist in modern OSes. With OS switching, at any given time, only one OS is active. When the user wants to switch to a different OS, the currently active OS is suspended to memory, and the target OS is resumed from a previously suspended state and becomes the new active OS. The OS switching approach promises native execution speed and full hardware access by all OSes, and offers relatively fast switching speed. Implementing OS switching requires only minor modifications to existing OS code, plus relatively simple code for controlling switching between OSes.

2. OS Switching with Suspend/Resume

2.1. Overview

An OS switching system has multiples OSes installed on the persistent storage (e.g. disks or flash drives). To differentiate from the term "guest OS" used in virtualization technology, we call each of these OSes a *tenant OS*. More than one tenant OSes can be loaded into disjunctive memory regions and can boot up one by one. However, at any time there is only one tenant OS actively running, and this OS is called the current *active OS*. The active OS owns completely the whole

1. Work conducted when author was with DoCoMo USA Labs. He can be contacted via email: jsun@junsun.net.
2. Work conducted when author was with DoCoMo USA Labs. He can be contacted via email: stevel@sklembedded.com.

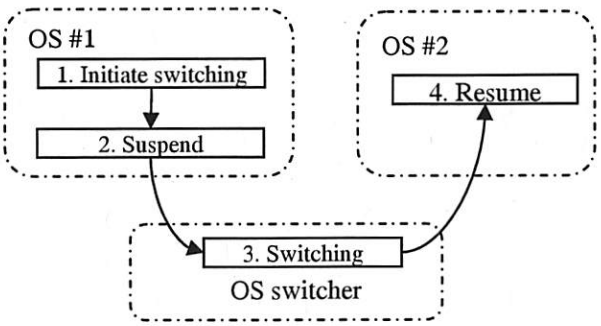


Figure 1. OS Switching Overview

system except for the portions of memory and disk reserved for other tenant OSes.

An active OS goes into dormant state through OS switching. OS switching is typically initiated by the end user (e.g., by pushing a switching button). It is also possible that OS switching is initiated through software triggered events.

After an OS switching is initiated, the current active OS (or outgoing OS in this context) performs necessary preparation, typically including saving necessary states for later resumption and putting hardware into a known and agreed-upon states for the next OS (or incoming OS).

Once the outgoing OS finished its preparation, an actual switch will happen. This step can be simply jumping to the resume path of the incoming OS. For OSes that support Memory Management Unit (MMU), however, this step may involve tearing down the outgoing OS' page mapping and setting up the new page table for the incoming OS.

The last step in OS switching is to restore incoming OS into an actively running state. This step involves retrieving states saved in system RAM and re-initialize hardware into a working state. Device drivers and application processes are re-activated, and system will continue to run from the state when the OS was last time suspended.

The four steps in OS switching is illustrated in Figure 1.

2.2. Suspend and resume in modern OSes

Most modern operating systems with advanced power management support a power-saving state where all hardware (including CPU and peripherals) are powered off except for the system RAM. In Windows this state is called *standby* state. Mac OS X calls it *sleep* state, while Linux refers to it as *suspend-to-RAM* (STR). When a computer performs suspend-to-RAM operation, the OS stops applications, drivers and kernel in order, and stores all necessary information in the RAM. The system then enters a low-power state while the RAM enters a low power self-refreshing state. Most of other hardware devices are turned off to save energy. When the system resumes, it retrieves operating state from memory and restores the whole system to the state when it was suspended.

Different OSes implement STR differently. The following text describes general steps involved in a typical suspend/resume process.

`Suspend Process:`

1. Suspend initiated
2. Applications are notified of the imminent suspend operation through callbacks. Certain applications may save data, or complete networking operations, etc.
3. Subsystems (such as file system, networking, daemons) are notified of the imminent suspend operation. For example, NFS domain may close its connection and save the connection information for later resumption.
4. The suspend routines in device drivers are called. Such suspend routines typically do two things: disabling its service to higher-level software and turn off the device (e.g., flushing and disabling DMA, disabling interrupts). Sometimes the suspend routine may also save certain information for later resumption.
5. Save system core state, including bus controller and CPU register states.
6. Turn off power to all hardware except system RAM, and enter sleep mode.

`Resume process:`

1. Resume is initiated through some pre-configured external events (pushing button, RTC timer expiration, etc). CPU control typically jump-starts from a pre-set address.
2. Restore CPU and system core states.
3. Invoke device drivers' resume() functions. The device driver resume functions typically enable the devices and make their services available for higher-level code.
4. Invoke the resume() functions of subsystems.
5. Notify applications of the resumption of operation. In Unix-like OSes, this can be achieved through signals.
6. System resumes full operation.

2.3. OS switching based on suspend/resume

If all tenant OSes support the suspend-to-RAM feature, we can re-use a large part of the suspend/resume code to implement OS switching. Conceptually, instead of turning off the power at the

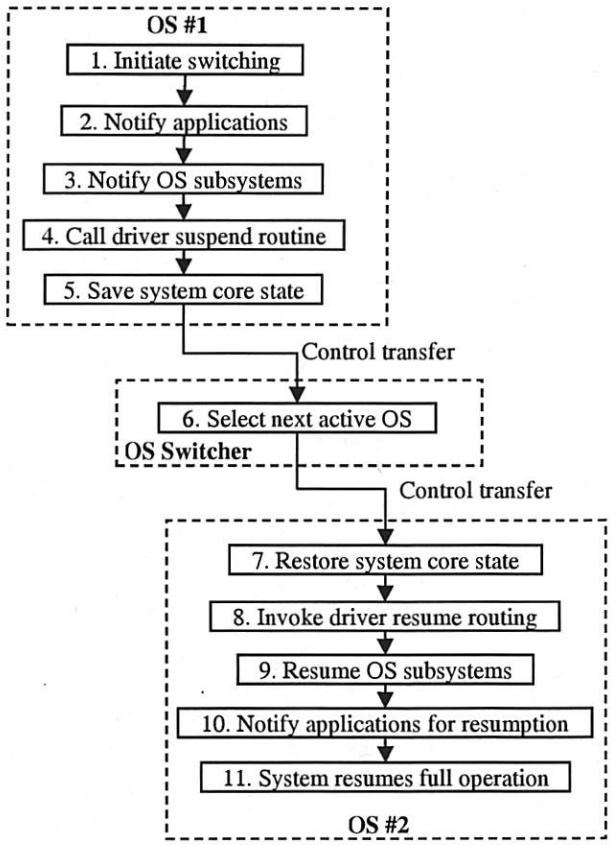


Figure 2. Flow of control in suspend/resume based implementation of OS switching

last step of the suspend process, we simply jump to the resumption path of the incoming OS.

Figure 2 shows the control flow of suspend/resume based OS switching.

If we are switching between two instances of the same OS with the same version, we are guaranteed that the hardware state at step 7 is exactly what the incoming OS expects. If we are switching between different OSes, however, the hardware states may be different from what the incoming OS expects. In that case, some form of "state adaptation" is needed (see section 2.5).

We implemented suspend/resume based OS switching in following environments:

- Linux-Linux switching on Sandgate 2P, an Intel PXA270 (ARM9) based handheld prototype device [10].
- Linux-WinCE switching on Sandgate 2P

Detailed implementation notes and performance data can be found in section 3. In the rest of this section, we will assume that there are two tenant OSes and discuss in general several other design issues.

2.4. Loading and booting of subsequent OSes

While the first OS in an OS switching system can be loaded and booted as usual (except that the loader and the OSes needs to be modified so that the OSes live in restricted memory regions), there are several design choices for loading and booting subsequent OSes:

1) The boot loader loads both OSes into RAM. One OS boots first and the second OS boots when it is selected to run for the first time.
2) The boot loader loads and boots the first OS. From the first OS an application loads the second OS. The second OS boots up when it is activated for the first time.
3) Boot loader loads and boots the first OS. From the first OS, an application installs the RAM content of the second OS previously captured when it went into suspended state.

Both methods 1) and 2) require special code that handles the booting of the second OS. Typically the loader and the OS mutually agree on the start-up system state. When for the first time we switch to the second tenant OS, the hardware state is usually different from what the loader would have set to. For example when Linux boots on ARM it expects MMU is turned off and first serial port is turned on. It also expects kernel command line arguments passed in through register r2. In order to boot the second OS correctly, we can either set the hardware states in the switcher so that they conform to the protocol, or we can modify the boot-up code in the second OS. We like the first approach as it is less intrusive in terms of changes to tenant OSes, and is more reusable when we switch among multiple different OSes.

Method 3) avoids the above problem. However, it requires a tool for capturing the memory content of the second OS in suspended state. In addition, the memory image, even when compressed, may be too large for systems where persistent storage space is scarce.

2.5. Switching

In theory, the actual control transfer is as simple as a jump instruction from the outgoing OS to the incoming OS.

In reality, this process is very complicated. The actual suspend process varies quite a bit for different CPUs, systems and OSes. In some OSes there are multiple suspend states. For example, Linux on VMPlayer has two suspend states corresponding to ACPI's S1 and S4 states [11].

Therefore, the first implementation decision is to choose a suspend path as the default OS switching path. Different suspend path puts hardware into

different suspend states and have different resume points. For example, Linux on VMPlayer support "standby" state, a shallower power saving mode where CPU context, including MMU and program counter, are preserved during suspend. Resume starts from the last instruction that puts the CUP into suspend state. By comparison, Linux on Sandgate 2P supports "memory" state, a deeper power saving mode where all CPU context are lost. The resumption point is remembered in a non-volatile register and CPU resumes in physical addressing mode.

As a result the switcher would need to manage all these differences and ensure the control transfer happen smoothly among tenant OSes. The OS switcher typically performs:

1. Saving any contexts that were assumed to persist during normal suspend but will be lost during OS switching.
2. Tearing down current MMU mapping and switching to physical addressing mode. If incoming OS resumes from virtual addressing mode, setup MMU mapping for the incoming OS.
3. Restoring any context for the incoming OS that were assumed to persist during normal suspend but was lost during OS switching.

3. Implementation and Performance

In this section, we describe our implementation of OS switching, and evaluate and analyze OS switching performance. As mentioned earlier, we have implemented two prototypes. In this section we focus on the Linux-WinCE Sandgate 2P prototype.

3.1. Loading and booting

In our Linux-WinCE Sandgate 2P prototype, we chose the first approach for subsequent tenant OS loading and booting, i.e., we modified the WinCE loader, eboot, to load both OSes into RAM. Eboot sets up the environment for WinCE to boot up first. When for the first time we switch from WinCE to Linux, the OS switcher sets up proper hardware state so that Linux can boot up successfully. Special setting include turning off MMU, turning on serial port, and preparing kernel boot arguments.

3.2. Switching on the Linux side

Our Linux kernel base is 2.6.16. Intel has supplied board specific support for Sandgate 2P. In this implementation, Linux supports two power saving states, "standby" and "memory". We decided to modify the suspend-to-memory execution path for OS switching purpose.

A side button on the prototype device is designated as the OS switching button. The keypad driver sends a signal to the APM daemon when a button pressing event is detected. Upon receiving this signal, the APM daemon performs the suspend process, including calling each driver's suspend() function. At the end of this process, instead of going into suspended state, the daemon jumps into the switcher and calls the switching function. In section 3.4 we will discuss in detail what the switcher does.

3.3. Switching on WinCE side

WinCE supports several power saving states including idle and suspend. Unfortunately the WinCE BSP we obtained from vendor does not fully support them. While some drivers have their own suspend/resume routines, some do not. In addition, there is no system-wide suspend/resume routines.

Our implementation effort starts with supplying those suspend/resume functions for various drivers including display driver. Similar to the Linux case, when the keypad driver detects an OS-switching button pressing event, it changes the system power state into suspend state, which starts the standard suspend process. The standard suspend process invokes the OEMPowerOff() function after all devices are suspended. The OEMPowerOff() function in turn invokes our real OS switching function.

3.4. Implementation of OS switcher

For practicality reasons, OS switcher is implemented inside eboot. Thus it also uses eboot's address mapping, which is different from either Linux's or WinCE's.

When we switch from Linux to WinCE, the switcher will save the CPU context, including MMU, general registers, system control registers, etc. It will then restore WinCE's CPU context. Since WinCE suspend/resume is not complete, the OS switcher performs additional saving and restoring for peripheral devices such as LCD, audio, etc.

When we switch from WinCE to Linux, the OS switcher performs similar steps. Again, for WinCE, the OS switcher saves additional context for peripheral devices. This saving is necessary as states for peripheral devices presumably will be altered once Linux becomes active.

3.5. Evaluation

In this section we present and discuss timing data for OS switching. The numbers presented in this section are obtained through instrumentation of Linux (2.6.16 kernel) and WinCE 5.0 source code. Since we

Table 1. Breaking down OS switching time (Linux)

Suspend Steps	Time Used (us)	Resume Steps	Time Used (us)
Freeze processes	8500	Thaw processes	3226
Device suspend	5845	Device resume	1384313
Other	32	Other	4796
Total	14377	Total	1392335

didn't have access to the full source code of WinCE 5.0, we could only measure suspend and resume time costs of major device drivers for WinCE side.

Table 1 breaks down the costs for switching out of and into Linux. Since process freeze/thaw and device suspend/resume times dominate the total cost, we omitted listing the costs of other individual steps. Note that the costs for freezing and thawing processes depend on those specific processes. In our experiment we only had a few basic processes running. As we can see from the table, the resume cost in Linux is much higher than the suspend cost, and this resume cost is dominated by the cost for resuming devices. Overall, the resuming process takes close to 1.4 seconds, and the total time for switching from a Linux OS to another Linux OS is slightly over 1.4 seconds (Linux suspend time plus Linux resume time).

Table 2. further breaks down the suspend/resume costs of individual Linux device drivers. Note that the resuming costs of the PCMCIA driver, the frame buffer driver, and the WiFi driver, dominates the total cost of device resumption.

On WinCE side, excluding GWES' (Graphics, Windowing and Events Subsystem) asynchronous handling of power on/off events, almost all the costs of OS switching is in device suspend and resume. Table 3 shows the suspend/resume costs of four drivers that were used on the WinCE side of the prototype: the drivers for display, touch screen, keypad, and audio. The total suspend time for the drivers is about 188ms, and the total resume time is about 341ms. We can thus infer that the time for Linux-to-WinCE switch is around 0.3-0.4 second, while the time for a WinCE-to-Linux switch is around 1.6 seconds.

Table 2. Suspend and resume cost of Linux device drivers

Driver	Suspend Time (us)	Resume Time (us)	Comment
ak4650-ts	2	3036	Touchscreen
ak4650-core	325	165	core for TS and audio
hostap_cs	1378	254231	Wifi driver
pxa2xx-pcmcia	26	769261	PCMCIA driver
pxa2xx-fb	3770	344926	Frame buffer
pxa2xx-ac97	328	12683	AC97 controller
Other drivers	16	11	
total	5845	1384313	

Table 3. Suspend/resume cost of WinCE device drivers

Device	Suspend Time (us)	Resume Time (us)
Display	153829	276103
Touchscreen	3439	4380
Keypad	3414	57316
Audio	27500	2862
Total	188182	340661

We also measured *user perceived switching time*, defined as the time from display going into blank in the outgoing OS, to the time display resumes in the incoming OS. It is measured as the time between the end of display driver suspend on one side, to the end of the display driver resume on the other side. In our setup, the user perceived switching time from WinCE to Linux is about 398.8ms, while the user perceived switching time from Linux to WinCE is about 328.3ms.

4. Related Work

The idea of using suspend/resume (and also shutdown/reboot) to support multiple OSes first appeared in a patent [9] by Shimotono. The patent generally assumes PC-like computing systems where BIOS performs the major part of switching. The outgoing OS completely shuts off the whole system and the incoming OS will start from reset state with a flag to indicate it is the resumption instead of a regular booting. From OS perspective there is no difference between a real suspend/resume and an OS switching.

Clearly this scheme does not work for non-PC systems where there is no BIOS standard and power management standard (such as APM or ACPI). In this paper we extend and broaden this idea to a more general OS switching approach where tenant OSes work cooperatively to time-share the same computing device. Shimotono's patent is a special case where all tenant OSes must suspend and shut off all hardware (even including CPU) completely before a switching can happen on the rebooting path in BIOS. In this paper we demonstrate that multiple OSes can suspend differently into different states and adapt through the OS switcher with a flexible scheme. Another directly related approach is the multi-boot approach [4]. Like OS switching approach, hard disks or permanent storage are partitioned among tenant OSes, and there is only one active OS at any time. Unlike OS switching approach, the active OS owns the whole system RAM instead sharing with other tenant OSes. However, OS switching offers much faster switching time.

Compared with virtualization technologies, OS switching lacks concurrency and hence is not suitable for application scenarios where multiple OSes need to run concurrently (for example, telnet from one tenant OS to another). In addition, OS switching depends on corporation among OSes and is consequently less robust against faulty OS implementations. On the other hand, OS switching offers native execution speed, which gives better performance than virtualization (especially traditional full virtualization). In addition, many application scenarios (such as multi-OS driver development) require native hardware access which is not possible in virtualization.

Compared with para-virtualization approaches such as Xen [5], OS switching requires less kernel modification. For example, our kernel patch for Linux/WinCE switching on ARM changes, excluding device driver changes, 26 lines of WinCE code and 139 lines of Linux, plus around 60 assembly instructions. In addition, OS switching only needs to change so-called BSP part of kernel, not as intrusive as other para-virtualization approaches. Because of this attribute, we are able to enable Linux-WinCE switching even though we don't have the full source of WinCE kernel.

The implementation of OS switching closely resembles cooperative VM approach in that all kernels have privileged access to the whole system and the cooperative relation among OS kernels. Cooperative Linux [12] modifies Linux kernel to run inside the host OS's kernel. The guest Linux kernel runs as a process on top of the host OS. MMU is time-shared between the host kernel and guest Linux kernel. Peripheral hardware access is virtualized through host OS's support. Jaluna's OSware [1] integrates two or more OSes and multiplexes hardware interrupts and CPU usage among them. Hardware resources are exclusively partitioned among OSes. Virtualized hardware access is possible if the owner OS exports the resource and the client OS has the virtual driver which knows how to talk to the owner OS. Compared with cooperative VM approach, OS switching approach trades multi-OS concurrency for implementation simplicity and full native access to hardware.

5. Summary and Conclusion

OS switching enables multiple OSes time-share the same computer in a cooperative manner. Its implementation typically reuses suspend/resume functionalities already existing in modern OSes and result in little modification to existing kernels. Compared with multi-boot approach, OS switching offers much faster switching time. Compared with virtualization approach OS switching offers simplicity, native execution speed and native hardware access.

In this paper we generalize the OS switching notion and present our study on its design, implementation, and performance. Despite some of its limitations we believe OS switching is a useful alternative to multi-boot approach and virtualization approach for many application scenarios where simplicity, performance, native hardware access and switching time are important. We would like to promote this approach and are hopeful to see wider applications of OS switching technology.

Acknowledgements

We would like to thank many of our colleagues for providing insights to our OS switching work, including Ken Ohta, Takehiro Nakayama, Jane Inamura, and Atsushi Takeshita. We would also like to thank Hiroshi Inamura for initiating the idea of putting multiple OSes on a mobile phone for improved system dependability.

References

[1] Jaluna. *Jaluna OSware*. Web site: http://www.jaluna.com.

[2] Intel and NTT DoCoMo, "Open and Secure Terminal Initiative (OSTI)," http://www.nttdocomo.co.jp/english/corporate/technology/osti/

[3] R. J. Creasy, "The origin of the VM/370 time-sharing system," *IBM Journal of Research and Development*, vol. 25, pp. 483-490, 1981.

[4] GNU. GNU Grub Project. Website: http://www.gnu.org/software/grub/.

[5] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the Art of Virtualization," in *Proceedings of the ACM Symposium on Operating Systems Principles*, October, 2003.

[6] Intel. *Intel Virtualization Techonology*. Web site: http://www.intel.com/technology/computing/vptech/.

[7] A. Whitaker, M. Shaw, and S. Bribble, "Denali: Lightweight virtual machines for distributed networked applications," in *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[8] VMWare. *VMWare home page*. Web site: http://www.vmware.com

[9] S. Shimotono, "Computer system, operating system switching system, operating system mounting method, operating system switching method, storage medium, and program transmission apparatus", US patent application number US20010018717A1, Aug 30, 2001.

[10] Sophia Systems. *Sandgate 2P reference design*. Web site: http://www.sophia.com/Products/SG2P.html.

[11] Advanced Configuration & Power Interface. "ACPI Specification". http://www.acpi.info/spec.htm.

[12] D. Aloni, "Cooperative Linux", in *Proceedings of the Linux Symposium (vol 2)*, pp.23-31, Ottawa, Ontario, July 21st-24th, 2004.

Cool Job Allocation: Measuring the Power Savings of Placing Jobs at Cooling-Efficient Locations in the Data Center

Cullen Bash and George Forman
Hewlett-Packard Labs, Palo Alto, CA 94304

Abstract

Data center costs for computer power and cooling are staggering. Because certain physical locations inside the data center are more efficient to cool than others, this suggests that allocating heavy computational workloads onto those servers that are in more efficient places might bring substantial savings. This simple idea raises two critical research questions that we address: (1) How should one measure and rank the cooling efficiency of different places in a data center? (2) How substantial is the savings? We performed a set of experiments in a thermally isolated portion of a real data center, and validated that the potential savings is substantial and therefore warrants further work in this area to exploit the savings opportunity.

1. Introduction

The total cost of ownership of a fully managed data center with a 1.3 megawatt cooling capacity is approximately $18 million per year (e.g. 100 fully loaded 13KW racks with 4000 1U servers) [7]. About 15% of the cost is for operation and maintenance of the environmental control system. This partly reflects that cooling resources are over-provisioned to cover worst-case situations. The temperature at the air inlet of all servers must be kept below a target threshold, $\leq 28^{\circ}$ C for example, even when all servers are 100% busy.

Local variations in airflow and server heat generation impact the efficiency of cooling different places within the data center. Air conditioning units on the periphery supply cool air into an under-floor plenum. The cool air is delivered to the room via ventilation tiles on the floor located in between the two rows of equipment racks. The equipment racks are oriented such that their air intakes are facing the "cold aisle" with the vent tiles. Hot spots in the top middle of the row result from recirculation of hot air exhausted on the opposite side of the server racks. The temperature of the exhaust air is primarily a function of equipment power consumption which is driven by server design and computational workload. Hot spots are a ubiquitous problem in air-cooled data centers, and drive the environmental control system to work much harder to ensure that no server is fed hot air (i.e. air at a temperature greater than the target threshold).

Related work [6,8,9,15] considered the placement of computational workload to alleviate these local hotspots and provide failure mitigation. Various algorithms have been developed to guide the placement of resources according to the external environment, but none yet have considered the varying ability of the air conditioning units to cool different places in the room. For example, a location might appear to be a good place because it is currently cold, but it may be difficult to cool, such as in a corner of the room in a location far removed from an air conditioning unit. Secondly, while simulations have been used to prove the concept of the various approaches, validation in a real data center under realistic workload conditions has not previously been attempted.

Section 2 describes a new practical metric to grade cooling-efficiency, which involves both the current air temperature and the historical ability of the computer room air conditioners (CRACs) to cool the location along with information about local airflow conditions. This metric can then be used to rank the different places in the data center, providing a preference for where to place heat-generating computational workload. The great complexity of managing a data center makes any additional considerations for cooling efficiency unwelcome. But the adaptive enterprise vision is that next-generation data centers will have management control software that will provide increased levels of automation and can more easily integrate cooling considerations into their policies. Of course, adding such software complexity to future data centers—as well as the research on how best to do it—is only warranted if the savings are sufficiently substantial. We address this strategic research question empirically:

We use our efficiency metric of Section 2 in a practical experiment described in Section 3 that measures the total power consumed by a thermally isolated portion of our data center under different control policies. The experiment assumes that computational workload, such as batch jobs, can be placed or moved within a data center based on cooling efficiency. Although this is not the practice today, it could be achieved easily enough by having job schedulers take a server preference list

Figure 1. Utilization of HP rendering service for 30 days (x-axis) over 345 servers (y-axis): white= busy.

into account when allocating large new jobs onto the servers, or else by future data centers that leverage Virtual Machine technology to dynamically migrate running jobs from one server to another in order to improve cooling efficiency.

The experiment results are described in Section 4. Briefly, we observed ~⅓ savings in the cooling power required, despite only having control of a fraction of the computers in the isolated data center. The ensuing discussion in Section 5 includes a translation of this savings into an estimate of the dollar savings for a modern, large scale data center. Depending on usage and other factors discussed in that section, it could easily exceed $1,000,000 savings per year.

We round out this introduction with a final item of motivation. The proposed savings depends considerably on the utilization of the data center, e.g. when the data center servers are ~100% busy or ~100% idle there is no flexibility about where to place workload. Thus, the potential for savings depends on the data center being only partially utilized a substantial fraction of the time. Although one cannot argue that this is the case in most data centers, we find various evidence that this is the case in at least some data centers:

1. Reports from the field indicate that many customer data centers run at fairly low utilization most of the time. Indeed, this has recently led to research and services in server consolidation via virtualization technology [12].
2. Anecdotal evidence of several academic batch job servers and our experience with those within HP Labs suggest that, although there are periods when all servers are continually busy (e.g. conference submission season), many other times the offered workload is sporadic.
3. As a final anecdote, we examined the utilization of the HP Labs movie rendering service used by DreamWorks in the production of the movie Shrek II, and again found substantial periods of middling utilization. Refer to the visualization in Figure 1, where a black pixel indicates a server was idle for an entire 5 minute interval, and is white otherwise. Over this 30 day period corresponding to Nov. 2004, we see many times when only a portion of the servers were busy.

Further analysis of this data, as well as additional discussion and color photographs, is available in the longer technical report version of this paper at: www.hpl.hp.com/techreports/2007/HPL-2007-62.html

2. Measuring Cooling Efficiency

Initial work regarding measurement and optimization of data center cooling efficiency was centered around the modeling and characterization of airflow and heat transfer in the data center [3,10,13]. The work relied upon numerical simulations to improve the placement of cooling resources via the manipulation of vent tiles, CRAC unit placement and server placement. Additional work has focused on the optimization of the fundamental equations of fluid mechanics and thermodynamics within racks [11] and data centers [14] to minimize a given cost function and improve operational efficiency. Although much progress has been made in this area, the modeling techniques involved are time consuming and have to be re-run as data center operation changes with time, either due to changes in workload distribution or physical configuration.

More recent work has focused on real-time control systems that can directly manipulate the distribution of cooling resources throughout the data center according to the needs of the computer equipment. One such system, called Dynamic Smart Cooling, uses a network of temperature sensors at the air inlet and exhaust of equipment racks [2]. Data from the sensors is fed to a controller where it is evaluated. The controller can then independently manipulate the supply air temperature and airflow rate of each CRAC in the data center. In order to accomplish this efficiently, the impact of each CRAC in the data center must be evaluated with respect to each sensor. The result of such an evaluation will define the "regions of influence" of each CRAC unit. This information can then be used to determine which CRACs to manipulate when a given sensor location requires more or less cool air. Such a system has been shown to operate much more efficiently than traditional control systems that contain sparse temperature sensing, usually only at the inlet of each CRAC, and rudimentary operating algorithms that do not consider local environmental conditions [2].

The regions of influence are defined with respect to a metric called the Thermal Correlation Index (TCI) shown in Equation 1. It quantifies the response at the ith rack inlet sensor to a step change in the supply temperature of the jth CRAC. TCI is a static metric based on the physical configuration of the data center.

Since it does not contain dynamic information, it can be thought of as the steady-state thermal gain at the sensor to a step change in thermal input at the CRAC. The regions of influence that are defined by the TCI metric are stable with time, but are functions of data center geometry and infrastructure (e.g. vent tile arrangement) as well as CRAC flow rate uniformity.

$$TCI_{i,j} = \frac{\Delta T_i}{\Delta T_{crac,j}} \quad (1)$$

The process by which TCI is evaluated can be performed numerically or in-situ in the data center with the deployed sensor network. In-situ measurements are more accurate while numerical simulations can be done off-line and enable parametric analysis.

Another attribute of TCI is that it describes the efficiency by which any given CRAC can provide cooling resources to any given server. We therefore use it in the development of a more general workload placement index that we term the "Local Workload Placement Index" described by Equation 2 as follows:

$$LWPI_i = \frac{(\text{Thermal Margin})_i + (\text{AC Margin})_i}{(\text{Hot Air Recirculation})_i} = \frac{(T_{set} - T_{in})_i + \sum_j \left[(T_{SAT} - T_{SAT,\min}) * TCI_i\right]_j}{(T_{in,i} - T'_{SAT,i})} \quad (2)$$

where the numerator quantifies the thermal management and air conditioning margin at sensor location i and the denominator quantifies the amount of hot air recirculation at the server (this is related to the Supply Heat Index described in [3]). Specifically, T_{set} is the desired computer equipment inlet temperature setpoint within Dynamic Smart Cooling, T_{in} is the current inlet temperature measured within the server or with an externally deployed sensor network, T_{SAT} and $T_{SAT,min}$ are the supply air temperature and minimum allowable supply air temperature of the air conditioning unit(s) respectively. Both are reported by the CRAC. The Thermal Correlation Index $TCI_{i,j}$ represents the degree to which CRAC j can provide cooling resources to the computer equipment at sensor i. Finally, $T'_{SAT,i}$ is the temperature of the air delivered through the vent tiles in close proximity to the ith server and is a strong function of the supply air temperature (T_{SAT}) of the CRACs that serve the region in which the ith sensor resides. As defined, the metric is a ratio of local (i.e. server level) thermal management and air conditioning margin to hot air recirculation and can therefore be used to gauge the efficiency of cooling resource placement and, by extension, workload placement.

Figure 2. Example server utilization, 8am burst jobs

3. Experiment Methodology

Fundamentally, any sort of workload might be placed so as to optimize cooling efficiency. We chose to focus on an opportunity that may be practical for widespread use in the near term: the placement of CPU-intense batch jobs. An obvious experiment scenario is to have jobs arrive occasionally, and to allocate each to the most cooling-efficient server available. It would remain only to choose job arrival rates and a distribution of job durations. One could then measure the power savings of cooling-efficient placement vs. today's cooling-oblivious placement. Though uncomplicated, this scenario is naïve. In batch processing systems it is common that a user enqueues a large number of jobs in a burst. For example, in the server utilization diagram in Figure 2, a large burst of jobs arrives at 8am, making all the servers go from idle (black) to busy (white). As each job completes, servers are kept busy by the supply of enqueued jobs. When the queue finally goes empty (~10am), each server runs its last allocated job to completion and then goes idle (the last being at 3pm). This type of pattern is evidenced repeatedly in the 30 day snapshot in Figure 1. Thus, in a practical deployment, the savings of cooling-aware placement will likely be realized only after the work queue is drained and servers begin to go idle.

The duration of this 'wind-down phase' can be substantial, especially if the variance in job lengths is large, as we often observe in practice. As a practical example, the job lengths in the NASA iPSC benchmark [5] have a coefficient of variation of ~350% (CV=std.dev/mean), and for a recent machine learning experiment by the second author, the CV was 130%. Thus, the placement of the last few long jobs determines which servers will remain busy long after the others have finished. It is here in the wind-down phase that we focus our experiments. We will compare today's cooling-oblivious, first-come first-served (FCFS) placement vs. a smart cooling-aware placement that puts the longer running jobs on the more efficient servers to cool, given the schedule shown in Figure 2.

In practical implementations, this could be achieved either by (1) having rough estimates of job lengths so that an efficient schedule can be devised, or (2) dynamically migrating long running jobs to the more efficient servers via virtualization technology, such as Xen. We initially attempted the latter, which is perhaps

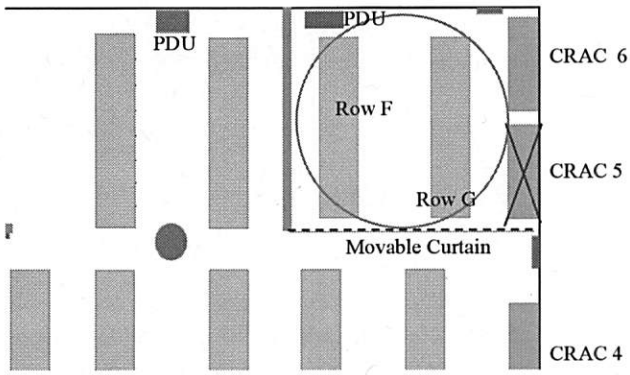


Figure 3. Experimental test bed.

more elegant because it can be difficult to obtain job length estimates. Unfortunately, due to ownership constraints, we could not get Xen installed on enough servers to make any significant impact on the room temperature, considering the many other computers present. Forced to resort to method (1), we developed a simple FCFS scheduler that placed the longest schedule on the most efficient server and the progressively shorter schedules on the less efficient servers in sequence. We used the pre-determined job lengths of 126 jobs from a previous experiment having CV 130%. This created the schedule shown in Figure 2.

But assuming one takes power savings seriously, there is another factor to consider: putting servers that are not being used into a low power state, e.g. shutting them off. This is simple enough to implement, and with quick hibernation available in future servers, it will become quite easy to effect. As we shall show, this complements efficient placement nicely, and used together, a great deal of power savings can be had.

To conclude, the experiment design follows a lesion study model, determining the power savings of cooling-efficient placement together with server shutdowns, as well as the marginal benefit of each technique by itself. The detailed protocol will be discussed after we introduce the test bed.

3.1 Test Bed

We are fortunate to have available to us a thermally isolated portion of an active data center at HP Labs, Palo Alto, depicted in Figure 3. We isolated the research area (upper right quadrant) via a heavy plastic curtain and closeable air baffles beneath the floor plenum. This area is cooled by two redundant Computer Room Air Conditioning units (CRACs), but for these experiments we turned off CRAC 5. Both CRACs 5 and 6 have two operational modes. One mode utilizes the facility's chilled water system to remove heat from the air via an internal heat exchanger while the other uses a vapor compression refrigeration system internal to the unit. When operating in the latter mode, power consumption of the unit can be directly measured and was therefore used throughout the experimental phase of this work. This CRAC was controlled via Dynamic Smart Cooling by using temperature sensors in the test bed. More modern Proliant-class HP servers have inlet temperature sensors built into each server.

We obtained control of 54 of the 76 NetServer LP2000r servers in the 6 racks in Row F (marked in Figure 3). Although other users had control of the remaining NetServers and many servers in the other row, we monitored their power consumption and discarded measurements affected by any substantial change.

3.2 Experiment Protocol

0. Rank the servers by their $LWPI_i$ value, computed from the temperature sensors and TCI efficiencies.
1. Determine the FCFS job schedules from the batch job predicted run-times, and place the longest running schedules on the most efficient servers.
2. At 8am, all servers go busy for two hours, giving the data center ample time to come to a thermally steady state, and reflects the situation after an arbitrary number of hours of being fully busy.
3. As each server completes between 10am and 3pm, it shuts down (simulating a low power mode).
4. As servers shut down and contiguous regions of servers around a sensor are all off, the acceptable temperature limit for that sensor is increased by about +5°C—enough to essentially remove it from control while still providing minimal cooling.
5. Measure the server and CRAC power consumption during the wind-down phase: 10am to 3pm (efficient placement has no effect when all servers are busy or all are idle).
6. On separate days, repeat the above experiment without server shutdowns, without efficient placement, and without either—for the baseline mimicking current behavior of batch services.

4. Experiment Results

Table 1 shows the experimental results in terms of average power consumed over the duration of the experiment and the savings with respect to the baseline setting. Power consumption of the servers in row F of Figure 3 is reduced by 30% when the test machines are shut down after their jobs have each completed. Naturally, the *server* power consumption is unaffected

by load placement. By contrast, power consumption of the air conditioning equipment is reduced by 8% via cooling-aware placement alone, 15% via shutdown alone, and 33% when both cooling-aware placement and shutdown are employed. Overall, the total power savings is reduced 32% when both techniques are used.

Table 1. Kilowatts consumed by each setting.

	Baseline	Placement	Shutdown	Both
Servers	16.2	16.2	11.4	11.4
% savings		0%	30%	**30%**
CRACs	25.2	23.2	21.4	16.9
% savings		8%	15%	**33%**
Total	41.4	39.4	32.8	28.3
% savings		5%	21%	**32%**

The savings afforded by cooling-aware placement of workloads without shutting servers down is due to the change in the distribution of heat that results in the reduction of recirculation of hot air into the inlet of the racked equipment. Recall that recirculation is a component of LWPI. This recirculation increases air conditioning costs, thus placement alone provides savings primarily in the cost to provision air conditioning resources. Shutting down machines, however, provides both savings at the power delivery level (i.e. power delivered to the computers) and the air conditioning level. The latter is due to the fact that the air conditioning system need not expend the energy to remove the heat formerly dissipated by the inactive computer equipment. When both placement and shutdown are used, added benefit is derived from the fact that clusters of machines in close proximity to each other are shut down as load is compacted to the most efficient places in the data center. These inactive clusters result in zones that can tolerate warmer air than active clusters and the cooling distribution can be adjusted accordingly (e.g. via Dynamic Smart Cooling) resulting in an additional 18% savings in the air conditioning costs from baseline over that achieved by shutting down machines without regard for placement (the shutdown scenario). Indeed, the air conditioning savings of including cooling-aware placement more than doubles the savings of shutdown alone.

5. Discussion

To help convey the practical impact of these results, we work through a simple computation to translate this savings into dollars, and then we discuss issues one may face in practical deployment. Finally, we give a remark on how difficult it is to perform this sort of research on a real, physical data center.

The results indicate that the application of job allocation based on environmental factors can significantly reduce the overall power consumption of the data center. As an example, if we consider a typical large-scale data center with a power consumption of 2.5 MW by the computational equipment (~190 13 kW racks) and a cooling load factor of 2.2 (defined as the ratio of the amount of heat being removed by the amount of power consumed by the air conditioning system to remove the heat), the total power consumption of the data center is 3.6 MW. (Note the load factor of 2.2 matches our experimental conditions and is a conservative assumption given that many data centers operate with load factors much lower than this—i.e. worse.) If we further assume that the data center is partially active 70% of the time per an analysis of data from Figure 1, and that the savings we observed in our experiment (32%) can be extended to the rest of the data center, at an energy cost of $0.15/kW·hr the energy savings will result in an operational savings of more than $1,000,000 per year. Naturally, a rough computation such as this is only an illustration.

Our experiments avoided several complications that may need to be surmounted for practical deployment, of course. One issue is that after servers have been shut down, they must be booted up again when new jobs arrive. It takes only a moment for Dynamic Smart Cooling to provide cooler air to such servers, but the delay of the reboot process is comparatively lengthy and undesirable. Future servers will have fast methods for low power or hibernation modes. Until available, one could trade off some cooling efficiency in order to avoid some of the boot-up delays by (a) never shutting off some of the most cooling-efficient servers, and/or (b) imposing a minimum idleness delay before shutting down any server.

We proposed to place workload either by requiring estimates of job length in advance (which can be difficult to obtain in most general settings), or else by migrating long-running jobs during the wind-down phase via virtualization technology. We believe this migration would be quite practical for most types of CPU-intense batch jobs, with only a sub-second suspension in computation. Note, however, that the job's memory needs to be migrated across the high-speed data center network. If the jobs are very memory intense, or if many migrations are requested in a short time window, the volume of network traffic may begin to pose a substantial cost and delay. Thus, practical controllers may occasionally need to temper their

eagerness for cooling efficiency in order to avoid network overload.

In our experiments, we only considered homogeneous servers, i.e. no matter which server is selected to run a job, the same amount of heat is generated. But over time, real data centers may accumulate a mixture of servers of different generations. Thus, optimal placement decisions may also need to take into account the differing amount of heat generated by different servers. And with widely different CPU speeds, the placement decisions will also affect how long the jobs take to complete. This leads to a complex area of optimization that mixes cooling efficiency considerations with traditional scheduling. Furthermore, economics may play a role: although it may be most efficient to run a user's jobs on a small set of old, slow servers that produce little heat, the user may be willing to pay more for a higher class of service that returns their results sooner at additional expense. These issues are beginning to be explored [4].

References

1. Andrzejak, A., Arlitt, M., and Rolia, J. "Bounding the Resource Savings of Utility Computing Models." HPL Technical Report, HPL-2002-339.
2. Bash, C.E., Patel, C.D., Sharma, R.K., "Dynamic Thermal Management of Air Cooled Data Centers", Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems, San Diego, CA, 2006
3. Bash, C. E., Patel, C., Sharma, R. K., 2003, "Efficient Thermal Management of Data Centers – Immediate and Long-Term Research Needs," International Journal of HVAC & R Research, Vol. 9, No 2.
4. Burge, J., Ranganathan, P., and Wiener, J. "Cost-aware Scheduling for Heterogeneous Enterprise Machines (CASH'EM)." HPL Technical Report HPL-2007-63.
5. Feitelson, D. G., and Nitzberg, B. "Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860." In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (Eds.), Springer-Verlag, 1995, Lect. Notes Comput. Sci. vol. 949, pp. 337-360
6. Moore, J., Chase, J., Ranganathan, P., Sharma, R. "Making Scheduling 'Cool': Temperature-Aware Workload Placement in Data Centers", USENIX 2004
7. Patel, C.D., Shah, A., "Cost Model for Planning, Development and Operation of a Data Center", HPL Technical Report, HPL-2005-107(R.1)
8. Patel, C.D., Sharma, R.K., Bash, C.E., Graupner, S., "Energy Aware Grid: Global Workload Placement Based on Energy Efficiency", Proc. of the ASME International Mechanical Engineering Congress and R&D Expo Nov 15-20, 2003; Washington D.C
9. Patel, C.D. "Smart Chip, System, Data Center – Dynamic Provisioning of Power and Cooling from Chip Core to the Cooling Tower", Temperature Aware Computing Workshop (TACS), Int'l Symposium on Comp. Architecture ISCA-2005, Madison, WI, 2005.
10. Patel, C. D., Bash, C. E., Belady, C., Stahl, L., Sullivan, D., "Computational Fluid Dynamics Modeling of High Compute Density Data Centers to Assure System Inlet Air Specifications," Proc. of the Pacific Rim/ASME Int'l Electronic Packaging Tech. Conf. and Exhibition (InterPACK), Kauai, HI, 2001.
11. Rolander, N., Rambo, J., Joshi, Y., Mistree, Y., 2005, "Robust Design of Air Cooled Server Cabinets for Thermal Efficiency," Paper IPACK2005-73171, Proc. of the ASME Int'l Electronic Packaging Tech. Conf. & Exhibition (InterPACK), San Francisco, CA, 2005.
12. Rolia, J., Cherkasova, L., Arlitt, M., and Andrzejak, A. 2005. A capacity management service for resource pools. In Proc. of the 5th Int'l Workshop on Software and Performance (Palma, Illes Balears, Spain, July 12 - 14, 2005). WOSP '05. ACM Press, NY, 229-237.
13. Schmidt, R., "Effect of Data Center Characteristics on Data Processing Equipment Inlet Temperatures," Paper IPACK2001-15870, Proc. of the Pacific Rim/ASME Int'l Electronic Packaging Technical Conf. and Exhibition (InterPACK), Kauai, HI, 2001.
14. Shah, A., Carey, V., Bash, C., and Patel, C. "Exergy-Based Optimization Strategies for Multi-Component Data Center Thermal Management," parts I and II, Papers IPACK2005-73137–8, Proc. of the ASME Int'l Electronic Packaging Technical Conference and Exhibition (InterPACK), San Francisco, CA, 2005.
15. Sharma, R.K., Bash, C. E., Patel, C. D., Friedrich, R.J., Chase, J.S. "Balance of Power: Dynamic Thermal Management of Internet Data Centers", IEEE Internet Computing, January 2005

Passwords for Everyone: Secure Mnemonic-based Accessible Authentication

Umut Topkara Mercan Topkara Mikhail J. Atallah
Department of Computer Science
Purdue University
utopkara,mkarahan,mja@cs.purdue.edu

Abstract

In many environments, a computer system is severely constrained to the extent that the practical input mechanisms are merely binary switches. Requiring the user to remember a long random bit string and to authenticate by entering each bit in the available binary input mechanism, is completely impractical. This paper deals with the question of authentication in such environments where the inputs are constrained to be yes/no responses to statements displayed on the user's screen. We present PassWit, a mnemonic-based system for such environments that combines good usability with high security, and has many additional features such as (to mention a few) resistance to phishing, keystroke-logging, and compatibility with currently deployed systems and password file formats (hence it can co-exist with existing login mechanisms).

1 Introduction

We present PassWit, an authentication system that can be used in environments where the input mechanisms are constrained to low bandwidth switches. For example, a disabled person may only be capable of yes/no responses to prompts from the screen (by different nods of the head, eye movements, or even by different thought patterns that are captured by a sensor). Alternatively, the user may not suffer from any impairment yet the environment precludes the use of a keyboard or keypad, as happens with tiny portable devices such as some of the smaller mp3 players, voice sensors at the doors of restricted-access areas, and hands-free situations such as construction work sites, operation of a motor vehicle, etc. Finally, a case can be made, in situations where shoulder-surfing is prevalent (such as in crowded cyber-cafes), for deliberately restricting the input to be a response that is hard to detect by a shoulder-surfer (e.g., left-click vs right-click).

PassWit enables users to achieve security of truly random passwords while authenticating themselves by just answering a series of "yes/no" questions. An important ingredient in our recipe is the use of a mnemonic that enables the user to produce a long enough (hence more secure) string of appropriate yes/no answers to displayed prompts (i.e., challenges). Each user is required to remember a mnemonic sentence of which they have several choices to pick from. Another important ingredient is the non-adaptive nature of these challenges – so they are inherently non-revealing to a shoulder-surfer or phisher. The mnemonic is a sentence or a set of words known only to the user and authenticating server (in the server they are stored in a cryptographically protected way rather than in the clear) – the users are never asked to enter their mnemonics to the system, they only use the mnemonic to answer the server's challenge questions. PassWit is safe against many attacks including shoulder surfing, phishing, and acoustic attack.

In PassWit, authentication can be achieved without requiring any special input device, or any computation at the client site. The authentication questions are designed in such a way that a short mnemonic sentence can encode a long password. There is no restriction on the size of the mnemonic sentence or the password, and the security (hence the length) of the passwords can be increased by requiring the user to remember more than one sentence.

Our design is compatible with pure text based interfaces as well as other media interfaces that can represent the text mnemonics. Our usage of text for mnemonics is not necessary but it is what we implemented for reasons of convenience and compatibility with existing login mechanisms; similar methods can be easily used in conjunction with speech, video, or pictures.

Moreover, using text for mnemonics (as opposed to pictures, video or audio) brings more flexibility to our system, since it is compatible with text consoles and LED screens. There is no language restriction for PassWit, it can be implemented for languages other than English.

2 Challenges and The Adversary Model

The primary challenge that we seek to overcome is to develop an authentication system which would be able to work in input-constrained environments. The ability to provide yes/no inputs makes it possible to transmit any random bit string but it does not help at all for remembering which bit string to transmit. A useful scheme needs to involve a mnemonic that makes it possible to securely remember a long random bit string, by remembering only a relatively short sentence.

A desirable authentication system should not require the user to carry an extra portable device (e.g. calculator) or even a paper and a pen to be widely applicable.

The password initialization and reset should be easy. In our case, initialization consists of asking the users on which topic they would like their mnemonic sentence to be, and later providing them alternative mnemonic sentences. Note that the users can use only yes/no answers to input their choice of mnemonic sentence.

We assume that all of the information used by the system during mnemonic creation is public and the adversary has equal (or more) computational power compared to our system.

The text passwords as well as the mnemonics in our system have to be secure against *dictionary attacks*. Mnemonics can not be a popular quote, or the lyrics of a well known song. We achieve this security by employing a mnemonic sentence generation method which was proposed as a part of a scheme for remembering conventional text passwords [16]. An adversary who has access to the passwords file (e.g., "/etc/passwd") does not gain any advantage when our system is used for authentication; our system does not weaken the security of existing authentication, it improves it by helping the users to have a truly random password.

A desired system should have resistance to attacks that involve capturing the user input for *replay attacks* (e.g. keystroke logging) or capturing the challenges (e.g. shoulder surfing). In our system the challenges and the user's responses are meaningless to an adversary, unless both are successfully captured at the same time.

Resistance to *phishing attacks* is a desired feature. Mnemonics are shared secrets between the user and the authenticating server and the users are never asked to enter their mnemonics to the system. The users can detect an adversary that does not know the shared secret. Otherwise an adversary does not gain any information about the password even if the user answers random phishing challenges.

Refer to [17] for a more detailed analysis of the challenges involved in building an authentication system for input constrained environments.

3 System Overview

We propose PassWit, an authentication system that is based on mnemonic passwords [16], whose details will be described in the following subsections, where $\mathcal{P}$ denotes the user's previously existing (and securely generated) password bit string (for now we assume $\mathcal{P}$ is 40 bits long, but we can accommodate any other length).

3.1 Password initialization step

1. The system generates a number of random sentences $s_1, \ldots, s_\lambda$ and displays them to the user. Each sentence has a length of μ words (not counting functional words such as "the", "a", "with"). In our implementation we used $\mu = 10$. For example, s_2 could come from tracing a random left-to-right path along the columns of Table 1, using some of the password bits to select one word from each column. In this case, 4 password bits are used per column and first column shows the bit string encoded by the words in the same row. For example, if $\mathcal{P} =$ 0101100101010011111101001000101010001101 then the resulting s_2 is *Angry union artists simply dismissed demand to forgive the laziness of the crazy mayor.* Each s_i is selected from a separate table like Table 1 which was derived from a different text source (e.g., sports news, stock markets, etc).

2. The user selects one of the above s_i's, suppose it consists of the successive words $m_1, m_2, \ldots, m_\mu$.

3. The column C_j from which word m_j was selected contains what we call the equivalence class (in that table) of the word m_j. We use r to denote the size of an equivalence class; in our example $r = 16$. The user does not need to memorize the equivalence class (only m_j needs to be remembered).

3.2 Authentication step

For $j = 1, \ldots, \mu$ in turn, the system asks the user , $\ell = \log_2 r$ questions ($\ell = 4$ in our example) about column C_j, as follows.

1. The system randomly permutes the entries of column C_j before creating the challenges at each session (which foils a replay attack). For the ith entry of C_j in the permuted order, let $b_{i,3}b_{i,2}b_{i,1}b_{i,0}$ be the binary representation of i. For instance last column of Table 1 might be permuted as {leader, senator, enemy, foe, king, queen, president, chairman, children, mayor, friend, ally, associate, assistant, manager, supporter}.

2. The system creates 4 sets Q_3, Q_2, Q_1, Q_0 such that the ith word of the permuted C_j is included in Q_k if and only if $b_{i,k} = 1$. For the permutation of C_{10} in the previous step, Q_0 would be {senator, foe, queen, chairman, mayor, ally, assistant, supporter}, and Q_1 would be {enemy, foe, president, chairman, friend, ally, manager, supporter}. Since the entry *leader* has index $i = 0$ in the permutation of this example session, it does not appear in any of the Q_k. See Figure 1 for the challenges of this session, which are displayed in random order (as opposed to alphabetical order) as CAPTCHAs for added security against sophisticated malicious software (the random re-ordering as well as the CAPTCHA representation are not needed if there is no threat of such an adversary).

3. For $k = 3, 2, 1, 0$ in turn, the system displays Q_k to the user who answers "Yes" if the mnemonic word m_j (corresponding to the current column C_j) is in Q_k, and answers "No" otherwise.

We note that (i) the user's answers uniquely identify to the server the mnemonic word in each column; (ii) the total number of questions is logarithmic in the size r of each column, so that password security can be increased by a factor of 2^μ by doubling the size of a column yet adding only 1 extra question per column (and, more importantly, without any increase in the size of the mnemonic, i.e., without further burdening the user's memory); (iii) a *shoulder-surfer* adversary sees the questions but not the user's yes/no answers (hence learns nothing); (iv) that a *keystroke-logger* sees the answers but cannot use them to authenticate itself or to obtain the passwords unless it can relate these answers to the challenges (which are preferably obfuscated as in the figure); (v) that a *phisher* adversary does not even know what questions to display, immediately alerting the user that something seriously phishy is going on (even if the phisher got a user to respond to very unfamiliar challenges, those responses are useless to such an attacker).

4 Implementation Details

We assume that the environment enables the user to read (or hear) the challenges displayed on the screen and the user can input the yes/no answers through a switch. The system includes a large set of tables, S, which are already populated offline. These tables, such as Table 1, are used for generating mnemonic sentences and challenges. Each table has a unique ID. Every table corresponds to a source sentence from a corpus, and these source sentences are stored in the first row of the table. Table 1 was generated using an example source sentence "Leading U.S. couturiers are strongly resisting pressure to regulate the thinness of the popular models." Every column in this table shows a possible candidate word set for replacing the original word in the first row (functional words are excluded).

4.1 Mnemonic Creation

At mnemonic-creation time, the system first generates a random password, $\mathcal{P}$, for the user (e.g. a random string of 40 bits), or the user's existing password is used. Next step is generating the possible candidates for mnemonic sentences that will encode this password.

If the source sentence has 10 words as in our example sentence, and each of these words have 16 alternatives; a random password chooses one word out of each of these 16 alternative words, hence encodes 4 bits per word, 40 bits in total.

The system selects the words that encode $\mathcal{P}$. This process generates one candidate mnemonic sentence per such table. All of the candidate mnemonic sentences encode the same $\mathcal{P}$.

At the end, the user is provided with a set of candidate mnemonic sentences to pick from and the random password, $\mathcal{P}$, to use in a keyboard setting if needed.

Mnemonic creation concludes with the user's selection of one of the candidate mnemonic sentences for remembering as a mnemonic.

Once the user selects which mnemonic sentence to use, the ID of the corresponding table that generated it is recorded in the least significant $\log_2 |S|$ bits of the salted hash of the password file entry.

Since we have many source sentences (say 1024 of them), the user can choose from 1024 different mnemonic sentences generated for each source sentence. However there might be psychological attacks [8] to such a flexible system; hence we advise only a small random portion of these possible mnemonics be given as choice to the users.

4.2 Mnemonic Usage

The mnemonic sentence is not stored in the system, instead the source table ID is stored in the salted password hash. The authentication involves a conversion of the yes/no answers of the user into a password.

We achieve this by generating the challenges in such a way that every yes/no answer narrows the search space by one-bit, similar to the idea behind the "20-Question Game". In our scheme, instead of looking for one object, we are searching for a password that is composed of concatenation of several substrings, each of which is encoded by a different word of the mnemonic sentence. Each mnemonic word is a member of an equivalence

	leading	U.S.	couturiers	strongly	resist	pressure	regulate	thinness	popular	models
0000	peaceful	viking	tailor	alarmingly	welcome	attempt	modify	rent	passive	queen
0001	thoughtful	romanian	cartoonist	hardly	agree	haste	alter	wisdom	inept	leader
0010	rich	city	beekeeper	suddenly	reject	duress	cement	culture	able	senator
0011	uninterested	rural	realist	simply	embrace	pressure	manipulate	education	dull	supporter
0100	provoked	irish	firefighters	warily	resist	demand	secure	diligence	hot	king
0101	angry	suburban	artist	doubtfully	renounce	bid	fix	weakness	skilled	ally
0110	outraged	texan	architect	remarkably	submit	call	quantify	salary	adept	foe
0111	neutral	aussie	police	again	honor	ultimatum	measure	pension	dormant	manager
1000	furious	canadian	cubist	blindly	recognize	struggle	forgive	thinness	crazy	friend
1001	poor	union	farmer	suspiciously	allow	operation	change	obedience	gifted	president
1010	average	british	fantasist	delicately	accept	order	limit	laziness	bright	enemy
1011	determined	european	developer	fiercely	surrender	imperative	throttle	spirit	witless	children
1100	strong	downtown	farmer	repeatedly	tolerate	hurry	harness	tenuity	exhausted	associate
1101	calm	urban	goldsmith	reluctantly	permit	insistence	deregulate	slenderness	talented	mayor
1110	silent	italian	musician	discreetly	refuse	ban	restrict	citizenship	clumsy	chairman
1111	ordinary	french	drivers	slowly	dismiss	decree	fiddle	discipline	sharp	assistant

Table 1: The mnemonic generation table for the sentence "Leading U.S. couturiers are strongly resisting pressure to regulate the thinness of the popular models." The order of words within a column is randomly determined.

class, and we need to ask several questions that will deterministically find the exact mnemonic word within a class. We ask $\log_2 r$, (e.g., 4), questions to determine one mnemonic word, where r, (e.g., 16), is the number of words in an equivalence class.

The key idea behind generating each challenge is very similar to the idea behind *non-adaptive blood testing* technique [9]. The area of combinatorial group testing concerns itself with performing group tests on subsets of a given set to identify defective elements in that set: A test for a subset tells whether that subset contains a defective element. If the set size is r and the number of defective elements is no more than d, then the goal is to pinpoint all the defective elements by making as few group tests as possible. The original problem was adaptive in the sense that test $i+1$ could be designed after the outcome of test i was known, thereby enabling a simple binary search for the defective element in the special case of $d = 1$. The non-adaptive version of the problem is when all the tests are done in a single round, with all the subsets to be tested determined in advance.

The analogy with our problem is as follows: For each mnemonic word m_j, the r "blood samples" are the r words in m_j's equivalence class. The mnemonic word is like the contaminated blood sample. The server presents the user with a subset of words from m_j's equivalence class, C_j, (possibly containing m_j) and the user is supposed to respond yes or no based on whether m_j is in that subset (i.e., whether that subset is "contaminated"). The server tests subsets in a manner that enables it to uniquely identify m_j, and then the server does a table lookup (local to the server) to derive the password bit string associated with m_j.

To prevent the adversary from learning anything by using the questions, it is imperative for the server to use a *non-adaptive* technique whereby *all* the questions have been pre-determined well in advance, as in non-adaptive combinatorial group testing. The questions are therefore independent of which item in the set is the "contaminated" one, and hence they reveal nothing to the adversary who sees them. Using adaptive group testing techniques (like binary search) for determining the questions would be lethal from the security point of view.

Our scheme will use $d = 1$, for which an $\ell = \lceil \log_2 r \rceil$ test non-adaptive solution is well known and in fact quite straightforward. We briefly sketch it, for the sake of making this paper self-contained. In what follows C_j is the equivalence class of word m_j, where $|C_j| = r$. (Recall the authentication step described in Section 3)

1. Let the words in C_j be listed in an order (which will be randomly changed at every authentication session) as $w_1, \ldots, w_\mu$.
2. For each word w_i, let the ℓ bit binary representation of i be denoted as the bit string $b_{i,\ell-1}, \ldots, b_{i,0}$.
3. For $k = 0, \ldots, \ell - 1$ in turn, the server's question Q_k is constructed as follows: Every w_j whose $b_{i,k} = 1$ is included in Q_k.

The server asks ℓ questions, and each question is constructed without any dependency on which element of C_j is the "contaminated" one, m_j. The server can easily determine m_j: It is the only word of C_j such that all of the Q_k's that contained it were answered with a "yes" by the user. Note that, this scheme is not restricted to inputting passwords using mnemonics, and it can be used to input plain text passwords when the challenges contain single ASCII symbols.

When the equivalence classes have a size of 16 as in our example, each challenge will have 8 words and the user will be asked 4 questions. An example question for finding the mnemonic word in the last column of Table 1 would be as follows:"Does your mnemonic sentence contain one of the following words?: { senator, foe, queen, chairman, mayor, ally, assistant, supporter }". The user answers 40 such questions in total (4 for each one of the 10 mnemonic words) with a "yes" or a "no" signal using the switch (equivalently with 1 for "yes" or 0 for "no").

After the answers are collected, the system extracts each password substring encoded by the mnemonic words and concatenates them in the order corresponding to the order of words in the source sentence to form the password $\mathcal{P}$. The hash of $\mathcal{P}$ with the salt is compared to the hash value kept in the password file (where the hash value is stored as in the regular UNIX password file). Note that the same password file can still be used with the ASCII passwords.

Our current password size of 40 bits falls short of the 52 bits commonly used in deployed systems, but we are confident that we will be able to exceed the 52 bits in the continuation and further refinement of this work. In the meantime, even in its present form our current implementation is suitable for use as a front-end to a 52-bit password system: We would use our system for entering 40 of the 52 bits, and the missing 12 bits would be handled as private salt in a similar fashion to what was described in [12] – by the front-end essentially trying all 2^{12} possibilities for the remaining 12 bits. The password file would stay the same as before our system was deployed, as would the password: We act only as a front end, when normal keyboard entry is either impossible or risky.

5 Related Work

Previous studies state the requirements for increasing the accessibility of electronic resources for the disabled users [1, 6, 5, 13, 2, 4] and suggest possible techniques to increase the bandwidth of input from these users [10, 7]. There is also a body of work for providing access to web through smaller devices which have limited input capabilities [3, 18]. These two research areas have a considerable overlap in the design requirements such as assumption of low input bandwidth, emphasis on usability, and the need for platform independence. Trewin discusses the overlap between the accessibility requirements for desktop browsers for Web, and the requirements for a usable Mobile Web in [18].

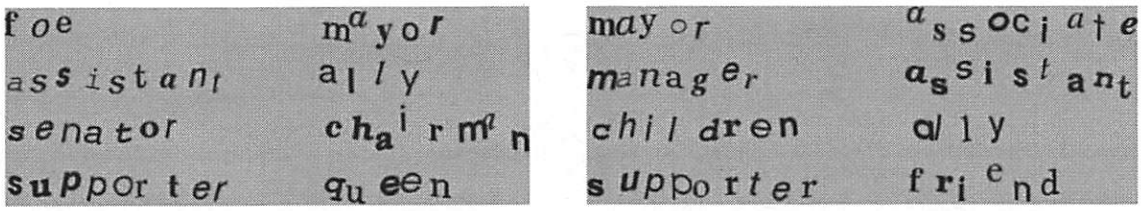
Mankoff et al. discuss the needs of motor disabled users for accessing the web in [13]. They study scenarios where the users control a switch through simple muscle movements such as raising an eye brow [10], or Brain Computer Interface (BCI) [19].

Pass-thoughts system [15] is based on recognition of unique brain signals send by the users. Thorpe et al. list the following set of requirements for an authentication system: i) changeability; ii) shoulder-surfing resistance; iii) theft protection (e.g. through acoustic attacks, or brute force attacks); iv) protection from user non-compliance; and, v) usability. The authors also present an authentication scheme that is solely based on training a user to think about the same idea (e.g. a place, a thing, or a melody), and recording the repeatable parts of the brain signal features extracted from this "pass-thought". Thorpe et al. reported that the BCI technology, that was available at the time of their study (September 2005), enabled the users to input approximately 25 bits per minute, which was not sufficient to provide enough bandwidth for the implementation of their sophisticated authentication scheme.

In 2004, Yan et al. conducted a controlled experiment to compare the effects of giving three alternative forms of advice about password selection [20]. The results of this study gave very valuable hints for designing a usable and secure authentication system: i) users have difficulty remembering random passwords (students in the group that was asked to write down their random passwords continued to carry the written copy for 4.8 weeks on the average.), ii) passwords derived from mnemonic phrases are indeed harder for an adversary to guess than naively selected passwords, iii) it is equally easy to remember strong passwords derived from mnemonic phrases and naively selected weak passwords.

Jeyaraman and Topkara proposed a system to increase the usability of text password authentication by automatically generating mnemonic sentences which help the users in remembering truly random passwords [11]. A more recently introduced mnemonic scheme for text password authentication by Topkara et al. [16] allows the users to maintain a multiplicity of truly random passwords, which are independently selected, by remembering only one mnemonic sentence. An adversary who breaks one of the passwords encoded in the mnemonic sentence does not gain information about the other passwords.

Note that in both schemes, [11, 16], the mnemonics are used as an aid to remember text passwords, whereas the current paper enables the use of the mnemonic sentence to serve as the password itself. In the current paper our main challenge is to construct an authentication mechanism that can work in restricted environments. We present a suggested mode of use for other mnemonic password schemes that use other media mnemonics including graphics, and audio besides text. The scheme in this paper provides resistance to phishing, to keystroke-



Q_0 ... Q_3

Figure 1: One of the possible set of challenges Q_0 to Q_3 that could be created for the last column of Table 1. The challenge words are presented in random order and in the form of CAPTCHAs for added security against sophisticated key-loggers. In the absence of such concerns the challenges can be displayed as text images in alphabetical order.

logging, to shoulder surfing as well as to dictionary attacks.

[14] presents a method that cleverly uses CAPTCHAs for assuring that an automated adversary will need more than a pre-determined amount of time to break a password through repeated login attempts.

6 Conclusion

We presented a password authentication system that is suitable for use in input-constrained environments, and that has many security and password-mnemonic advantages over existing keyboard-based schemes. Because of its compatibility with existing systems (to which it can act as a front-end), it can be used in an intermittent fashion alongside these existing systems: A user may prefer to use the normal keyboard entry most of the time (e.g., at home and in the office) but occasionally switch to using our system in certain situations, such as when the user fears the presence of shoulder-surfers or surveillance cameras, or has a temporary wrist injury that prevents the use of a keyboard, etc.

7 Acknowledgements

Portions of this work were supported by Grants IIS-0325345 and CNS-0627488 from the National Science Foundation, and by sponsors of the Center for Education and Research in Information Assurance and Security. The authors would like to thank the four anonymous reviewers for their helpful feedback and suggestions.

References

[1] Information technology accessibility and workforce-division, section 508: The road to accessibility, 1998.

[2] Australian banker's association inc., guiding principles for accessible authentication, Accessed on 4 December 2006.

[3] W3c mobile web initiative, Accessed on January 10, 2006.

[4] W3c web accessibilty initiative, Accessed on January 10, 2006.

[5] BBC-NEWS. Most websites failing disabled, Published on 2006/12/05.

[6] BROWN, C. Assistive technology computers and persons with disabilities. *ACM Communications* (1992).

[7] COPESTAKE, A. Applying natural language processing techniques to speech prostheses. In *Working Notes of the 1996 AAAI Fall Symposium on Developing Assistive Technology for People with Disabilities* (1996).

[8] DAVIS, D., MONROSE, F., AND REITER, M. K. On user choice in graphical password schemes. In *13th USENIX Security Symposium* (2004).

[9] DORFMAN, R. The detection of defective members of large populations. *The Annals of Mathematical Statistics* (1943).

[10] GRAUMAN, K., BETKE, M., LOMBARDI, J., GIPS, J., AND BRADSKI, G. Communication via eye blinks and eyebrow raises: video-based human-computer interfaces. *Universal Access in the Information Society* (2003).

[11] JEYARAMAN, S., AND TOPKARA, U. Have the cake and eat it too – infusing usability into text-password based authentication systems. In *21st Annual Computer Security Applications Conference* (2005).

[12] MANBER, U. A simple scheme to make passwords based on one-way functions much harder to crack. *Computers and Security* (1996).

[13] MANKOFF, J., DEY, A., BATRA, U., AND MOORE, M. Web accessibility for low bandwidth input. In *5th International ACM Conference on Assistive Technologies* (2002).

[14] PINKAS, B., AND SANDER, T. Securing passwords against dictionary attacks. In *ACM Computer and Security Conference* (2002).

[15] THORPE, J., VAN OORSCHOT, P. C., AND SOMAYAJI, A. Passthoughts: Authenticating with our minds. In *Workshop on New Security Paradigms* (2005).

[16] TOPKARA, U., ATALLAH, M. J., AND TOPKARA, M. Passwords decay, words endure: Secure and re-usable multiple password mnemonics. In *ACM Symposium on Applied Computing* (2007).

[17] TOPKARA, U., TOPKARA, M., AND ATALLAH, M. J. Passwords for everyone: Secure mnemonic-based accessible authentication. Tech. Rep. CSD TR #07-008, Purdue University, 2007.

[18] TREWIN, S. Physical usability and the mobile web. In *International Cross-disciplinary Workshop on Web Accessibility* (2006).

[19] VAUGHAN, T., HEETDERKS, W., TREJO, L., RYMER, W., WEINRICH, M., MOORE, M., KUBLER, A., DOBKIN, B., BIRBAUMER, N., DONCHIN, E., WOLPAW, E., AND WOLPAW, J. Brain-computer interface technology: a review of the second international meeting. *IEEE Transactions on Neural Systems and Rehabilitation Engineering* (2003).

[20] YAN, J., BLACKWELL, A., ANDERSON, R., AND GRANT, A. Password memorability and security: Empirical results. *IEEE Security and Privacy* (2004).

Virtually Shared Displays and User Input Devices

Grant Wallace and Kai Li
Department of Computer Science,
Princeton University, Princeton, NJ 08540

Abstract

This paper proposes making displays and input devices as first-class citizens in a networked system environment for collaborative applications. The paper describes a virtually shared model that enables users to use remote displays as extensions of their local displays and to allow multiple users to use multiple cursors and keyboards to input and control shared applications and their windows simultaneously. We have implemented a prototype system and deployed it to three DOE Fusion control rooms. The implementation performs well on today's hardware and our user feedbacks show that such a paradigm can substantially improve information sharing.

1. Introduction

We are moving into a new era of computing in which computers and networks are becoming ubiquitous. One of the associated challenges is to achieve seamless communication and visualization, i.e. to enable users with heterogeneous computing devices to communicate with each other and to visualize each other's information effortlessly and seamlessly.

We advocate that displays and user-input devices should be first-class citizens in a networked environment. They should be connected such that display information can be moved from one display to another seamlessly for collaboration purposes, functionally similar to connecting to external physical displays. Users should be able to view shared displays as extensions of their own displays, ideally independent of platforms, operating systems and applications. A scientist could walk into her colleague's office to show results by simply dragging the plots from her laptop onto a colleague's desktop display. After their discussion, the display information on the colleague's display may evaporate. In a collaborative group setting, multiple users can display, input and control information on a shared display simultaneously (Figure 1).

Current operating systems and window systems, however, have limitations to supporting such scenarios. They were designed for single-user use cases in a less network-centric era. A fundamental assumption has been that display devices and user-input devices are not first-class citizens in a network system environment. As Gettys pointed out in his paper on SNAP computing [Gett05], *"Today's computing mantra is 'One keyboard, one mouse, one display, one computer, one user, one role, one administration'; in short, one of everything. However, if several people try to use the same computer today, or cross administrative boundaries, or change roles from work to home life, chaos generally ensues."*

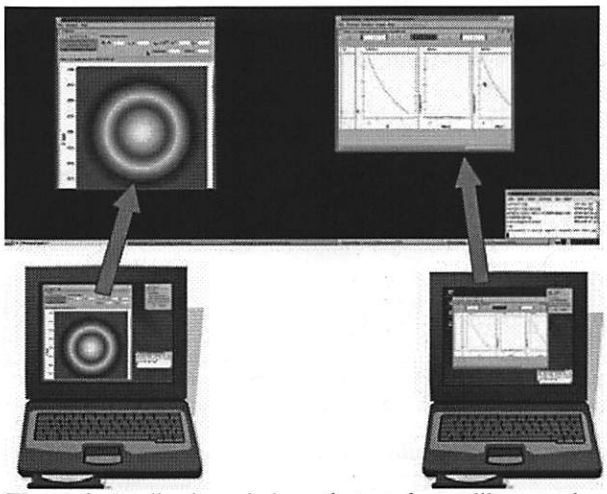


Figure 1: Application windows that can be readily moved or duplicated to other screens, and simultaneous multi-user input form the core of a collaborative display environment.

This paper proposes virtually shared display and user-input abstractions to create network enabled displays and input devices. The abstractions can be used to conveniently design systems to implement seamless information sharing for multi-user collaborations.

We have designed and implemented a shared display system by using the proposed abstractions and by leveraging VNC and x2x. We have released the implementation to the public domain and deployed our shared display system to three DOE fusion control rooms for their production use. The users' feedback shows that the proposed abstractions are indeed useful and shared displays can substantially improve information sharing in control room environments. Our performance evaluation shows that the implementation can provide interactive frame rates and impose reasonable network and computing resource requirements.

2. Virtually Shared Displays: Model and Abstractions

We propose a *virtually shared model* for displays and user-input in a networked environment. The goal is to conveniently connect components across a network so that peer participants can share application windows and provide user input across physical devices. There are several requirements to support this goal:

- Share display information at the granularity of application windows.
- Allow multiple users to input and control applications simultaneously.
- Support sharing in a many-to-many fashion.
- Be operating system independent and application transparent.

The main abstractions for this model are networked user-input devices, networked application windows, and networked displays. These components form communication links such that input-devices connect to applications, and applications connect to displays.

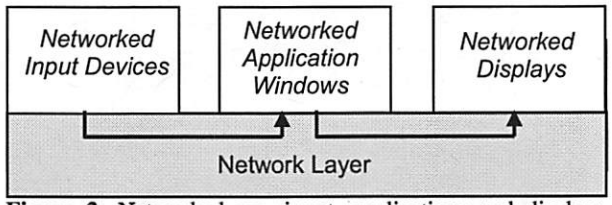


Figure 2: Networked user-input, applications and displays are the basis of a shared display environment.

A ***networked window*** is an abstraction associated with an application that sends graphical outputs and receives user inputs across the network. Networked windows must support several operations related to these tasks.

For graphical output, a networked window must support the operations of *replication*, *migration* and *orphaning*. Replication allows network windows to be shown on multiple networked displays simultaneously. Migration allows a networked window to be moved from one networked display to another. Orphaning allows an application to continue running even when no network display resources are available.

A networked window must also be able to receive input from networked input-devices. If the networked application is multi-user aware it can allow for simultaneous input; otherwise, it can serialize multiple input streams that it receives.

These networked window operations permit a variety of use cases such as collaborative sharing (replication), mobile computing (migration), and display composition to view high-resolution spanned windows.

A ***networked display*** is an abstraction that implements a display surface for computer systems in a network to display application windows. The main difference from the traditional display abstraction is that it allows for multiple focused windows per display to support multiple simultaneous networked input devices. The main operations of a networked display are *attach/detach window*, *attach/detach cursor*. Attach window is the operation that allows a networked application window to begin showing content on the networked display. Attach cursor is the operation that allows a networked user-input device to provide input to the networked display. Note that only networked windows on a display can be seen or controlled remotely by others. In other words, the networked or non-networked property of a group of windows serves as the protection domain for information sharing in this paradigm.

A ***networked input device*** is an abstraction that virtualizes a physical user-input device for composing systems in a network. The goal of the abstraction is to treat input devices also as first-class citizens in a network to enable building flexible multi-user systems.

Traditionally, user input is forwarded to the focused application by the display server or window manager. In our model, the networked input devices can instead make direct connections to their focused networked applications. In this case, the networked display still tracks cursor-to-window focus relationships but, instead of forwarding input, it provides the URL address of the networked application when focus relationships are established. This can provide for a direct communication model with better security and performance.

The main operations of a networked input device are *attach/detach* to or from a networked application. This abstraction allows dynamic binding with networked window abstractions. Thus, they can be used to compose an implementation that allows for multiple simultaneous inputs.

3. Previous Work

It is desirable to leverage previous work in this area in order to establish a collaborative display system for fusion scientists without excessive engineering effort. The final system must meet the four requirements listed in section 2: many-to-many, window-granularity sharing, with multi-user input across varied hardware platforms.

The X windows system provides a network-based display protocol that makes it possible to connect to remote display servers. However such connections can only be established at application startup and so dynamic replication or migration of windows is not supported [Gett04]. Several X11-based proxy servers have been created such as SharedX, Xmove and DMX to allow display redirection. However, they also require applications to connect at startup and aren't cross-platform compatible. These limitations do not lend themselves to the type of ad-hoc and dynamic collaboration required.

A number of collaborative systems provide a one-to-many sharing paradigm such as LiveMeeting, NetMeeting, Remote Desktop, Citrix and WebEx. These systems allow one person at a time to share display information and provide input. They, unfortunately, do not support many-to-many or peer-to-peer type sharing where multiple participants simultaneously share windows and provide user input.

Another class of collaborative display systems is based on VNC (Virtual Network Computer). VNC uses a pixel-based approach to replicate all desktop pixels from one computer to another [Rich98]. The advantage of this type of approach is its support for dynamic, cross-platform sharing. One variant of VNC called MetaVNC [Sato04] allows remote desktop windows and local windows to appear side-by-side on the desktop. This is accomplished by making the background of the remote desktop appear transparent. The main drawback of MetaVNC and other VNC implementations is that sharing is done at the granularity of the desktop. When connecting to a remote MetaVNC server, all desktop windows will be shared. We would like to restrict the sharing granularity to the window level for privacy reasons.

THINC is another virtual display system that allows networked desktop sharing. It is implemented at the device driver level and as such can support dynamic sharing and achieve good performance [Bara05]. However, because it operates at the device driver level, it does not track application window boundaries and so doesn't support window granularity sharing. Additionally it currently only has a Linux implementation.

Previous work on supporting multiple simultaneous user input has been done in the Computer-Supported-Cooperative-Work (CSCW) community. It focuses on multi-user computer-human interaction [Cars99]. Related work has also been done in the Single-Display Groupware community to look at multi-user interaction on a shared display [Myer99]. These classes of research have typically looked at human-interface needs and application support for collaboration rather than at systems level requirements. They have not looked at OS- and Window Manager- level support for multiple cursor interaction and in particular do not address multi-user interaction on legacy systems and applications.

A recent effort has been made on a multi-pointer X11 server [Hutt06]. This server allows multiple mice to be plugged into the same computer to create multiple cursors and simultaneous interaction. This work is very relevant and could be extended in the future to support a networked input abstraction.

4. Design and Implementation

In implementing the proposed collaborative display abstractions, our approach is to leverage existing software components where possible. VNC can form a good basis for display sharing because it supports cross-platform sharing; however, it must be modified to add support for window granularity and many-to-one sharing. Additionally, x2x forms a good basis for creating networked user input. It is X11 based, but can easily be supported on Windows systems using Cygwin. It, however, must be modified to support capturing and distinguishing multiple simultaneous user input.

Leveraging these software systems, we have designed and implemented the three networked components as described below and shown in figure 3. We will discuss the implementation of these components in the remainder of this section.

- ***Networked application windows*** – supported with a modified VNC server that allows capturing and sharing pixels at window granularity.
- ***Networked user input*** – supported with a modified version of x2x that can generate distinct input from multiple users.
- ***Networked displays*** – supported with a modified VNC viewer that can display windows from multiple servers and with a modified window manager that supports simultaneous multi-cursor input.

As mentioned earlier, VNC is a pixel-based protocol that allows easy cross-platform sharing. However, VNC only provides functionality to share the entire desktop, not individual windows. This has undesirable consequences for both privacy and utility in collaborative display systems. Privacy issues arise because users typically have some content they want to share, such as a data graph, and other content they want to keep pri-

vate, such as their email. Additionally, utility is limited when an entire desktop is shared because application windows from different collaborators cannot be placed side-by-side for comparison and discussion.

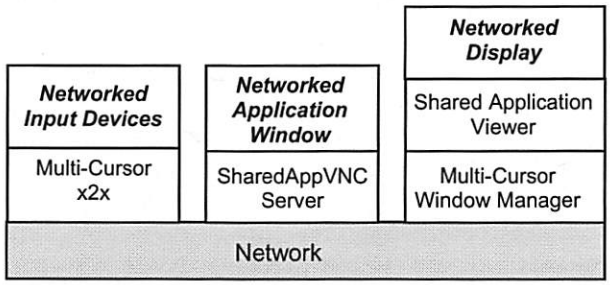


Figure 3: Shared display system architecture using virtually shared display and input device abstractions.

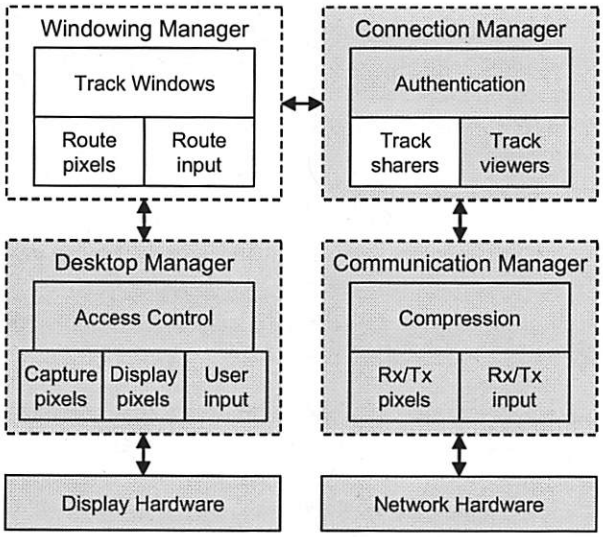


Figure 4: Original VNC provides display and management of pixels at the granularity of the desktop. We extend this by adding a windowing manager to allow window-granularity sharing between multiple sources.

To overcome these limitations we created an extended VNC protocol and implementation as depicted in Figure 4. Original VNC implements the gray components including the capture, transmission and display of pixels at the granularity of the desktop. In addition, it handles one-to-many connection management, which allows one presenter to share to many viewers. Our implementation has added the functionality represented by the white boxes. A windowing manager is added to maintain window-level knowledge, such as the location, size and relationship of all desktop windows. This permits the sharing of certain windows and exclusion of others. In addition, we extend the viewer connection manager to handle multiple simultaneous connections. This allows many-to-one sharing where many users share content to the same viewer, such as a display wall.

The multiple shared windows on the viewer are each placed in their own frame and so look identical to native windows of the viewer display. The shared windows include all parts of the application display including menus. The shared windows can be rearranged independently of their positions on the originating computer, and so can be placed anywhere on a networked display. This allows for easy side-by-side comparison of shared windows. If a shared window becomes occluded on its originating server, it will stop sending display updates for that region and that viewer content will remain static until the occlusion is removed. Our implementation, SharedAppVNC, is released to the public domain (http://shared-app-vnc.sourceforge.net/).

Allowing multiple users to simultaneously interact on the collaborative display was the second priority in our prototype system. Current windowing systems only have data structures supporting a single cursor, so to accomplish simultaneous interaction we created a specialized X11 window manager [Wall04].

The window manager renders multiple cursor arrows on the screen by drawing small 16x16 pixel windows and utilizing the XShape extension to make their shape identical to a normal cursor. Each multi-cursor is rendered with a unique color to easily distinguish it from the others. Cursor events are sent to the window manager using a modified version of x2x which packs the cursor id into 3 unused bits in the XEvent *state* field. This allows the distinction of 8 unique cursors. The cursor id allows the window manager to maintain multi-cursor state information including the current location, focused window and activated control keys.

When the window manager receives a multi-cursor event, it applies that cursor's saved state to the system cursor and then resends the event through the normal event loop (figure 5). This process essentially time-slices the system cursor between the multi-cursors. The time-slicing will be imperceptible to simultaneous users because user input such as typing or dragging is of low bandwidth compared to the system and display update response. Also, we suppress the z-order and window decoration changes that normally happen during a keyboard focus event. This makes keyboard focus changes unnoticeable to users. Window decorations and z-order are only changed when a multi-cursor establishes focus on a window. At that point the window is decorated with the cursor's color to designate the focus relationship where input will be directed.

Operations that involve mouse dragging are the one instance where events cannot be effectively interleaved. For this situation we allow users to obtain an exclusive lock for dragging. By pressing the shift button while dragging, all other multi-cursor events are suppressed. The other cursors will appear as an X until the drag operation is completed.

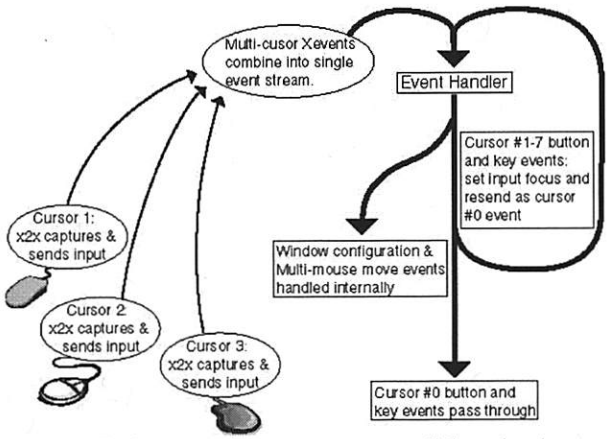


Figure 5: Our prototype system accomplishes simultaneous multi-user input by time-slicing the system cursor provided by a standard operating system.

The multi-cursor window manager is implemented for X11 displays; a release version based on IceWM is available at http://multicursor-wm.sourceforge.net/.

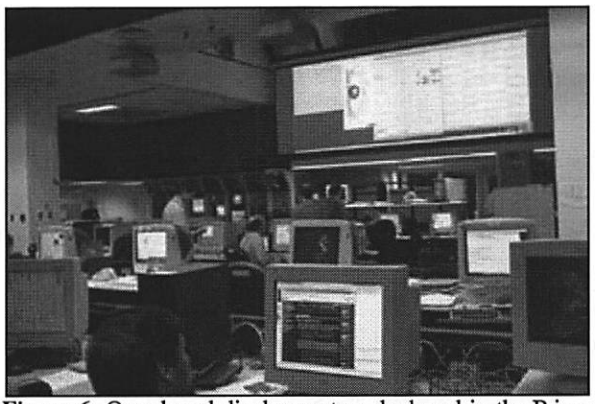


Figure 6: Our shared display system deployed in the Princeton Plasma Physics Lab control room.

Our initial system has been evaluated by several fusion facilities and is currently part of the production environment in the control rooms of General Atomics in San Diego, and the Princeton Plasma Physics Lab. This system has changed how the display walls in those rooms are used. Instead of just showing pre-determined content, the shared displays are now a dynamic forum for user discussion. In a typical discussion, 2-5 users will push application windows onto the wall and compare their results side-by-side. We anticipate that future stages of our system will also integrate remote users into the collaborative discussions.

5. Evaluation

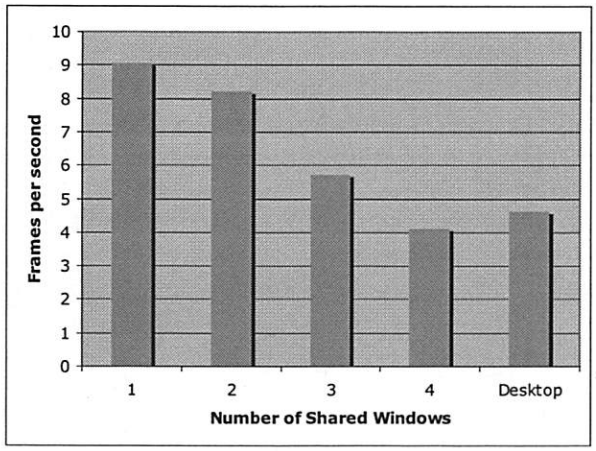
We have evaluated our system on its ability to provide adequate frame-rate and response time. In a Fusion control room, users typically share a few types of data including 2D plots, animated 2D plots and video clips from the Fusion engine camera. We test the system with similar workloads.

In the first experiment, since video clips are the most resource-intensive to share, we measured the frame-rates achievable when sharing multiple video clips from one workstation to the shared display. For this experiment we used a 3.2GHz Pentium 4 with 2GB of memory running Fedora Core 5 for both the scientist workstation and the computer driving the shared display. They were connected by 100Mbit Ethernet and the scientist workstation had a resolution of 1280x1024. The scientist workstation ran 4 video windows, each occupying about 25% of the screen (640x480).

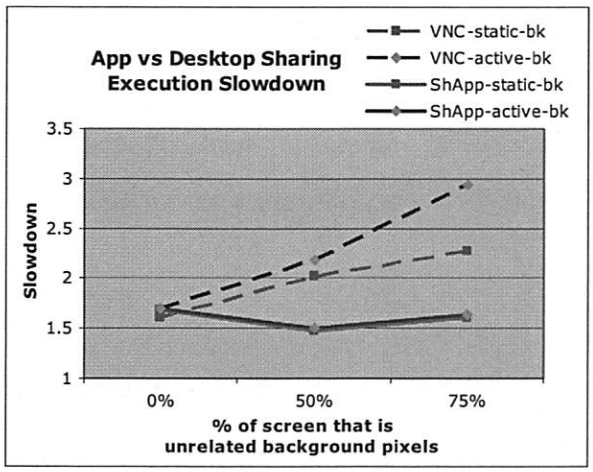
We measured the video frame-rate achievable while sharing 1, 2, 3 or all 4 windows simultaneously and compare that with sharing the entire desktop with unmodified VNC (graph 1). Sharing one or two windows achieved a frame-rate of about 9 and 8 fps respectively, still satisfactory for typical control room simulation videos. As additional videos were shared, frame-rates fell off as expected, giving about 4 fps with 4 windows shared. This validates our assumption that sharing smaller portions of the screen should provide better performance. For comparison, sharing the whole desktop using normal VNC gives about 4.5 fps, about equivalent to sharing all 4 windows with SharedAppVNC.

In addition to adequate frame-rate, interactive applications must provide good response time to be useable. To measure response time we made a trace of a user interacting with a 2D dataset application. We then used a robot tool to replay the user input from a remote display and measured the execution time. For comparison we measured the execution time of the trace on the local computer without sharing; this formed our base execution time, normalized to 1. We measured the execution time of the trace application occupying 100%, 50%, or 25% of the originating screen; this corresponds to background pixel activity occupying 0%, 50% or 75% of the screen. We ran two cases, one with the background pixels static and the other with the background pixels active. The active background case con-

sists of an image slide-show with image transitions every 1 second. The results are shown in graph 2. For single-window sharing using SharedAppVnc, the slowdown is typically a factor of about 1.5 for both the static background content and the slide-show background (solid lines graph 2). This is as expected because individual window sharing can ignore the background pixel data. We expect a slight performance improvement with smaller windows, a trend we see initially when the window reduces to 50% of the screen.



Graph 1: Frame-rates achieved using SharedAppVNC to share multiple video windows. SharedAppVNC achieves higher frame-rates by limiting the screen-area shared.



Graph 2: Execution slowdown of an application shared using VNC or SharedAppVNC. The window size and background pixel activity is varied. SharedAppVNC keeps relatively constant performance by avoiding sending background pixels.

We also measured the slowdown using normal VNC sharing the entire desktop (dotted lines, graph 2). Unlike with window-sharing, desktop-sharing slowdown is affected by the background pixel activity. For a full screen trace-window, the relative slowdown is equivalent to that of SharedAppVnc – about 1.5. For smaller trace-window sizes, the relative slowdown for VNC increases because VNC must also transfer the background pixel activity to the remote client.

These experiments validate our experience that SharedAppVnc can achieve better performance when smaller areas of the screen are shared, and that the performance slowdown for the remote user is tolerable.

6. Conclusion

We have proposed a virtually shared model for networked displays and user-input devices. Using this model we have implemented a prototype system, released it to the public domain, and deployed it into the control rooms of two DOE fusion research labs where it has been incorporated into daily production use.

Some quotes from scientists indicate their appreciation for the collaborative system. "[Previously] everyone had their own screen, or hardcopy. To collaborate, they usually looked over someone's shoulder. [The collaborative software] allows easy side-by-side comparisons of data from different people...[and] lets scientists make connections and correlations between displays and data sets that would be difficult without the wall."

These positive results and feedback encourage us to continue future research and enhancements to shared display environments such as implementing a direct communication model between networked components so that latency can be improved and permissions settings supported.

7. References

[Bara05] Baratto, R., Kim, L., and Nieh, J., "*THINC: A Virtual Display Architecture for Thin-Client Computing*", ACM Symposium on Operating Systems Principles (SOSP 2005).

[Cars99] Carstensen, P., Schmidt, K.," *Computer supported cooperative work: New challenges to systems design*". In K. Itoh (Ed.), Handbook of human factors, 1999.

[Gett05] Gettys, Jim, "*SNAP Computing and the X Window System*", Linux Symposium, July 2005.

[Gett04] Gettys, J., Packard, K., "*The (Re)Architecture of the X Window System*", Linux Symposium, July 2004.

[Hutt06] Hutterer, P., MPX, http://wearables.unisa.edu.au/mpx

[Meng94] Menges, J., Jeffay, K., "*Inverting X: An Architecture for a Shared Distributed Window System*", Workshop on Infrastructure for Collaborative Enterprises, 1994.

[Myer99] Myers, B. and Stiel, H., "*An implementation architecture to support single-display groupware*", CMU Technical Report, CMU-CS-99-139, 1999.

[Rich98] Richardson, T., Stafford-Fraser, Q., Wood, K., Hopper, A. "*Virtual Network Computing*", IEEE Internet Computing, 2(1), Jan/Feb 1998.

[Sato04] Satoshi, U., *MetaVNC*, http://metavnc.sourceforge.net

[Wall04] Wallace, G., Bi, P., Li, K., Anshus, O., "*A Multi-Cursor X Window Manager Supporting Control Room Collaboration*", Princeton University, Computer Science, Technical Report TR-707-04, July 2004.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to *;login:*, the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications: see http://www.usenix.org/membership/specialdisc.html

For more information about membership, conferences, or publications, see http://www.usenix.org.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at http://www.sage.org.

Thanks to USENIX & SAGE Corporate Supporters

Ajava Systems, Inc.

Cambridge Computer Services, Inc.

cPacket Networks

DigiCert® SSL Certification

EAGLE Software, Inc.

FOTO SEARCH Stock Footage and Stock Photography

Google

GroundWork Open Source Solutions

Hewlett-Packard

IBM

Infosys

Intel

Interhack

MSB Associates

NetApp

Oracle

Raytheon

Ripe NCC

rTIN Aps

Sendmail, Inc.

Splunk

Sun Microsystems, Inc.

Taos

Tellme Networks

UUNET Technologies, Inc.

VMware

Zenoss

ISBN-13: 978-1931971539
ISBN-10: 1931971536